

高性能并行珠玑

多核和众核编程方法

[美] 詹姆斯·赖因德斯 (James Reinders) 吉姆·杰弗斯 (Jim Jeffers) 等编著

张云泉 袁良 贾海鹏 李士刚 曹婷 译

High Performance Parallelism Pearls
Multicore and Many-core Programming Approaches

**High Performance
Parallelism Pearls**

Multicore and Many-core Programming Approaches

James Reinders, Jim Jeffers



机械工业出版社
China Machine Press

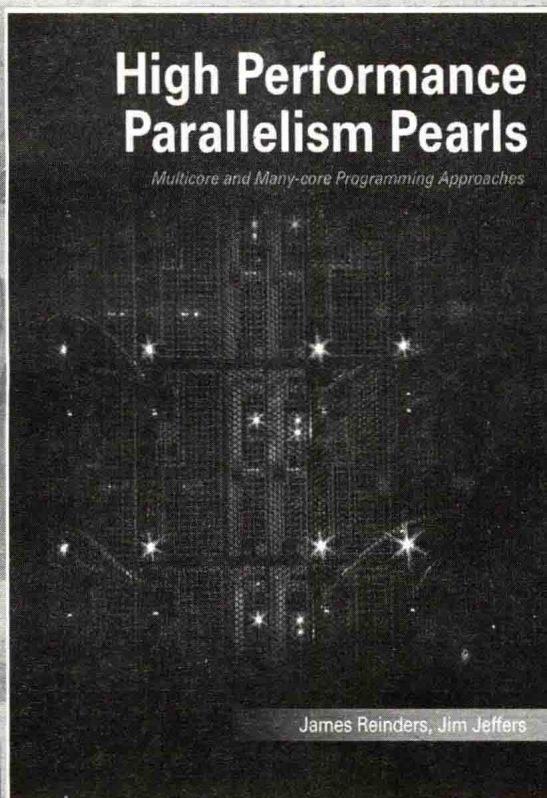
计 算 机 科 学 丛 书

高性能并行珠玑

多核和众核编程方法

[美] 詹姆斯·赖因德斯 (James Reinders) 吉姆·杰弗斯 (Jim Jeffers) 等编著
张云泉 袁良 贾海鹏 李士刚 曹婷 译

High Performance Parallelism Pearls
Multicore and Many-core Programming Approaches



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

高性能并行珠玑: 多核和众核编程方法 / (美) 詹姆斯·赖因德斯 (James Reinders) 等编著; 张云泉等译. —北京: 机械工业出版社, 2017.9

(计算机科学丛书)

书名原文: High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches

ISBN 978-7-111-58080-5

I. 高… II. ①詹… ②张… III. 微处理器—程序设计 IV. TP332

中国版本图书馆 CIP 数据核字 (2017) 第 234758 号

本书版权登记号: 图字 01-2015-2810

High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches

James Reinders and Jim Jeffers

ISBN: 978-0-12-802118-7

Copyright © 2015 by Elsevier Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

Copyright © 2017 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR, Macau SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由 Elsevier (Singapore) Pte Ltd. 授权机械工业出版社在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 出版及标价销售。未经许可之出口, 视为违反著作权法, 将受民事及刑事法律之制裁。

本书封底贴有 Elsevier 防伪标签, 无标签者不得销售。

本书由 Intel 的技术专家撰写, 全面系统地讲解在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上进行并行处理和编程的方法和技术, 展示了利用 Intel 处理器或其他多核处理器系统计算潜力的有效方法。全书包括大量来自多个行业 and 不同领域的并行编程例子。每章既详细讲述所采用的编程技术, 同时展示了其在 Intel Xeon Phi 协处理器和多核处理器上的高性能结果。大量案例显示的“成功经验”不但展现了这些强大系统的主要特征, 而且展示出如何在这些异构系统上保持并行化。

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 谢晓芳

责任校对: 李秋荣

印刷: 北京瑞德印刷有限公司

版次: 2017 年 11 月第 1 版第 1 次印刷

开本: 185mm×260mm 1/16

印张: 25.25

书号: ISBN 978-7-111-58080-5

定价: 119.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

近十年，我国高性能计算机的发展突飞猛进，最近“天河二号”连续六次夺得全球超算 TOP 500 排行榜第一，标志着我国硬件的发展已经达到国际先进水平。然而，在高性能软件方面，我国至今未获得过 Gordon Bell 奖这一超算应用大奖，并行应用的发展整体上仍然落后于发达国家。我国高性能计算学术界和产业界早已充分认识到这一问题，多年来已设立众多相关研究项目并取得了一系列重要研究成果，与国外先进水平的差距也在逐步缩小。

本书由来自工业界和学术界的众多一流科研人员共同编写，不仅包括大量不同的实际并行应用的优化经验，例如 N 体问题、量子化学、流体力学以及深度学习等，还包括一般性方法，例如 SIMD、OpenCL、文件系统以及并发调度等。这些方法多为最新硬件架构上的研究成果，例如 GPU 和 Xeon Phi。此外，本书配备大量程序代码实例，方便读者掌握相关技巧。总之，本书作者通过自己在高性能计算领域多年的实际编程和优化经验，选取最为关键和实用的优化技术进行讲解，可供相关领域的科研人员参考。

我们在阅读了本书英文版后，感觉本书有以下几个特色：首先，本书由长期在一线从事高性能应用开发和优化的专家编写，具有较强的实用性；其次，本书覆盖许多不同的高性能计算应用领域，范围十分广泛；再次，本书大部分应用涉及在最新 Intel Xeon Phi 协处理器上的优化设计，可借鉴性较强；最后，本书还包含大量通用优化技术的介绍，优化方法也具有广泛的适用性。

基于以上原因，我们组织了本书的翻译工作，具体分工如下：张云泉负责推荐序、前言、作者简介以及第 7、12、14、28 章的翻译，袁良负责第 2、6、9、17、23、26 章的翻译，贾海鹏负责第 8、11、13、16、18、20、22 章的翻译，李士刚负责第 3、4、5、21、25、27 章的翻译，曹婷负责第 1、10、15、19、24 章的翻译，最后由张云泉研究员审读全书译稿，以保证翻译的前后一致性。

由于时间仓促，加上书中某些术语目前没有统一译法，所以我们对一些术语采取了保留其英文名称的方法。对书中翻译方面的错误和不妥之处，恳请广大读者不吝批评指正。

——张云泉

能够为本书写推荐序，我感到非常荣幸，因为 Intel Xeon Phi 协处理器是一个激动人心的“计算神器”，欧洲核子研究中心（CERN）和高能物理（HEP）界都利用它来满足巨大的计算需求。当然，世界上还有许多科学和工程项目会因此受益。本书也包含了大量可以帮助程序员和科学家学习如何充分利用多核和众核处理器的实例。

堆积如山的计算需求：科学年正在形成

为了能够达到粒子加速器前所未有的高能量范围，CERN 在 21 世纪初就开始为大型强子对撞机（LHC）做准备。

然而，这个项目已经不仅仅是构建粒子加速器。四个新的实验已经形成，科学家们也都在努力工作。ALICE、ATLAS、CMS 和 LHCb 实验有数千个物理学家参与。无论是探测器的设计和建设，还是大型软件重建框架的设计和开发，都需要处理拍字节（PB）级的实验数据。这一切努力都是在寻求突破性的发现。

开放的标准

对于工业界，标准非常重要。投资开放标准一直是 CERN 的目标。我们发现 Intel 的理念也是一样。在与 HEP 界数百个科研机构 and 实验室的合作下，我们决定建立基于商用 x86 服务器的 LHC 计算网格（LCG）。在 2001 年，CERN 成立了开放实验室，用于调研可能与 LCG 有关的新兴技术。从一开始，Intel 就作为工业合作伙伴加入其中。在开放实验室里，我们进行的是领先于时代的研发工作，其结果可能是巨大的成功，也可能是惨败。这里曾经是且目前仍是一个非常真实的研发实验室！

对开放标准的浓厚兴趣，也驱使我们鼓励 Intel 将 Intel Xeon Phi 协处理器移植到 Linux。Intel 成功地实现了一个基于 Linux 的非常稳定的系统，并开放源代码，以鼓励创新。

热衷于众核架构

十年前，Intel Xeon 处理器系列很快就演变成了强大的多核处理器。这是 HEP 界的天赐良机，因为每个物理事件（碰撞后所有粒子的状态）都独立于所有其他物理事件，所以大量的事件可以采用细粒度并行进行计算。在开放实验室，我们一直努力确保每一代新系统都能获得最佳的吞吐量。

在努力满足不断增长的计算需求的过程中，Intel 所取得的发展为我们提供了新的选择。在 2008 年，我们非常热衷于 Intel 的“Larrabee”项目。虽然最初被宣传为图形处理器，但其本质上其实是世界上首个片上 SMP 集群。在这个设备上运行标准程序或程序内核应该是很容易实现的。当时没有硬件，只有如 Michael Abrash 的“First Look at the Larrabee New Instructions”这样的文章，在这篇文章中，Michael Abrash 描述了标准 x86 指令集的一组非常复杂的向量扩展。在 Intel 实验室的 Pradeep Dubey 及其团队的帮助下，我们通过模拟代码片段看到了可喜的成果。可以并行地处于活跃状态的数百个 x86 线程引起了我们的兴趣。

而常用的编程模型、语言和工具也将被证明仍具有巨大的价值。

Intel 最初的性能目标主要集中于单精度 (SP) 浮点计算, 这非常适合图像处理, 但对于计算物理是远远不够的。我们的评论以及对于高性能双精度 (DP) 浮点计算的需求, 在 Intel Xeon Phi 产品中得到了体现。

在 2009 年, 除了 Intel 外, 我们是第一个拿到了 Knights Ferry (KNF) 卡的团队, 可以想象我们当时激动的心情。尽管现在把我们的所有论文成果结合在一起为时尚早, 但 Intel 已经致力于这样的工作。与此同时, 我们想要将一些 LHC 实验移植到 KNF 上。重离子实验 ALICE 和其他一些重离子实验的科学家们相互合作, 开发了新版软件 “Trackfitter”。新的代码根据传感器的坐标 (传感器在给定的粒子碰撞中会发光) 重构了整个轨迹, 也就是重构了粒子的轨迹。在开放实验室, 我们已经把它移植到 Intel Xeon 处理器上, 因此只用了几个小时就将其移植到 KNF 上了。数天的工作就可以获得这样出色的成果, 我们非常激动。

Xeon Phi 的诞生: 众核, 卓越的向量 ISA

我荣幸地收到邀请, 于 2010 年 5 月在国际超级计算大会 (ISC) 上分享我们早期的工作成果。在这次会议上, Intel 数据中心及互联系统事业部负责人 Kirk Skaugen 宣布 Knights Corner 为第一个产品级众核设备。不久之后更名为 Intel Xeon Phi, 预示着它将是系列产品, 就像 Intel Xeon 一样。

你可能会问: 为什么我们对此如此热衷? 因为我们在 Intel Xeon Phi 的设计中看到了一个卓越的向量指令集架构 (ISA)。通过将向量屏蔽寄存器与向量数据寄存器分开, 该架构能够采用更优的方式处理程序中的数据流和控制流。Intel 一直对业界的高昂积极性感到高兴, 他们最近发布的 AVX-512 指令集将推动这个向量架构也加入他们的多核处理器。

同时, 我们也对众核的设计很感兴趣。程序的指令级并行 (ILP) 有限, 因此将它们并行运行在功耗只有几瓦的内核上效果会更好。我们与探测器模拟软件 Geant 4 的开发者一起将数百个线程运行在一个 Xeon Phi 协处理器上, 执行整个探测器模拟进程来演示这种方法的优点。未来将告诉我们小内核、大内核或者两者的混合, 哪个是最好的选择。我们的目标是对所有可能都做好准备。这个众核方案所取得的令人鼓舞的成果使我们想要尝试将这种方法应用到一个单纯的处理器上, 而不是一个协处理器上。我们非常期待基于使用下一代 Intel Xeon Phi (也称为 Knights Landing (KNL)) 的单机众核处理器的可引导系统。

学习具有高可扩展性的并程序序设计

Intel Xeon 和 Intel Xeon Phi 系列内核数量的激增, 为 HEP 界带来了福音。高能物理科学具有天然的并行性。真正令人兴奋的挑战是扩展到众核和向量化的过程。CERN 开放实验室已经为此做出了贡献。近十年里, 开放实验室在一系列的研讨会和高校中教授向量化和并行化, 目前已经有超过 1000 名学生。我很高兴能够为大家介绍这本书, 希望越来越多的读者能够学习如何使用这个堪称 “业界标准” 的现代微处理器来提高程序的性能。

未来需求增长: 编程模型的重要性

LHC 在 20 年内仍将继续运行。我们还雄心勃勃地计划提高能量和亮度, 这必将导致事件的复杂性增加, 从而提高对计算的需求。因此, 我们必须通过向量化获得更高的效率。

重建物理事件的挑战在于如何捕捉每个交互的所有细节。在 Intel 的帮助下, 我们实施

“Geant V”项目来完成软件的向量化，以迎接这些新的挑战。受经验的启发，例如本书中所列的那些经验，在不久的将来我们很可能实现更高性能的解决方案。

本书将使为 Intel Xeon Phi 产品开发高层并行性（包括最优编程）变得更加简单。Intel Xeon 和 Intel Xeon Phi 系列之间的通用编程方法对整个科学和工程领域都是有利的，相同的程序可以实现多核和众核的并行可扩展性和向量化。LHC 还将运行很长时间，希望 Intel Xeon Phi 系列处理器能够在此期间不断优化来满足我们大量的计算需求。

——Sverre Jarp

CERN 荣誉成员

本书囊括了 69 位作者的 Intel Xeon Phi 协处理器并行编程经验，他们将处理器和协处理器的性能发挥得淋漓尽致。其中讨论了并行编程中许多关键的挑战和技术，并给出了令人激动的成果。大多数章节展示如何良好地进行扩展和向量化，这将有利于在多核处理器和众核 Intel Xeon Phi 协处理器上获得更好的性能。其他章节揭示如何在提供了通用编程模型的系统下利用由 Intel Xeon 处理器和 Intel Xeon Phi 协处理器组成的新异构系统。书中还提供了关于部署、管理、监控与运行这些新异构系统和集群的专家建议。

来自 61 个内核的灵感：编程新纪元

对我们来说，比 Intel Xeon Phi 协处理器的成功更引人瞩目的是 Intel Xeon Phi 协处理器对并行编程的激励。这个协处理器开启了编程的新篇章。而在多达 61 个内核上的并行编程似乎远比在 4 个或者 8 个内核上的并行编程更有吸引力。它激发了人们将并行编程技术首次应用于一些应用程序以及改进已有的并行应用程序的兴趣。它激励人们研究真正可扩展的并行编程，而不仅仅是在只有少量并行（比如四核处理器上）时取得的尚可（有时甚至微小）的性能提高。

Intel Xeon Phi 协处理器为并行化带来了变革，为在其中探索的人们带了巨大的机遇。在这个过程中，我们不需要新的编程模型、新的语言或者新的工具。本书提供的并行编程工作和思想，描述了如何将旧的技术应用到新的异构编程平台上。这将帮助我们挖掘这一平台的巨大潜力。

我们非常感谢各位作者。各章的作者致力于在这个令人鼓舞的强大设备上编程。工作之余，他们为我们详述自己的工作，以便使我们学习他们的成功经验。我们希望你能够从中受益，并在这个并行计算的新时代获得成功。

致谢

本书的完成首先要感谢为此付出努力的软件开发工程师们，他们在工作之余与我们分享经验。本书浓缩了各位作者的成果。他们的名字列在其所写章节的开头，“作者简介”中有关于他们的简要介绍。我们要感谢所有作者坚持不懈的努力与理解。

感谢我们共同的朋友 Sverre Jarp 在“推荐序”中分享他独特的见解，感谢 Joe Curley 鼓励我们完成这件几乎不可能完成的事情。

James Reinders 感谢妻子 Susan Meredith，她的支持对于本书的完成至关重要。同时，James 也感谢女儿 Katie 和儿子 Andrew 一直以来的大力支持。最后，James 要感谢合著者和朋友 Jim Jeffers，感谢他又一次成为完美搭档。

Jim Jeffers 感谢妻子 Laura 一如既往的支持和鼓励。Jim 感谢孩子（包括孩子们的配偶）Tim、Patrick、Colleen、Sarah、Jon，尤其是刚出生的孙女 Hannah，他们无时无刻不在鼓舞着他。最后，Jim 非常感谢合著者和朋友 James Reinders，感谢他的专业技能和指导，感谢他坚定地恪守承诺使这本书从概念变成现实。

感谢 Joe Curley、Bob Burroughs、Herb Hinstorff、Rob Farber 和 Nathan Schultz 提供的支持、指导和反馈。

感谢整个 Morgan Kaufmann 团队的辛勤工作，包括与我们直接合作的三个人：Todd Green、Lindsay Lawrence 和 Priya Kumaraguruparan。

许多同事提供了信息、建议和想法。当然，还有很多人直接或间接地提供了帮助，对此我们深表感激。感谢所有帮助过我们的人，并对我们忘记提到名字的所有人表示歉意。

感谢所有人。

——Jim Jeffers

James Reinders

Intel 公司

2014 年 11 月

作者简介

High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches



Mustafa AbdulJabbar 是沙特阿拉伯阿卜杜拉国王科技大学 (KAUST) 极限计算研究中心的博士生，他致力于研究大规模算法优化 (如 FMM)，并对弥补基于 RMI 的执行模型与分子动力学和流体力学中的现实应用程序之间的差距感兴趣。



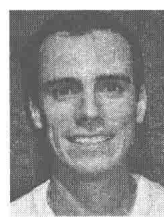
Nikita Astafiev 是俄罗斯 Intel 公司 Numerics 团队的一名高级软件工程师。他致力于研究高度优化的数学函数。他感兴趣的关键领域包括浮点误差自动分析和底层优化。



Jefferson Amstutz 是美国 SURVICE Engineering 公司的一名软件工程师。他探索可视化交互和高性能计算，为美国陆军实验室的应用程序提供支持。他致力于解决领域中各种基于物理的模拟问题，例如弹道脆弱性分析、无线电频率传播和柔体模拟。



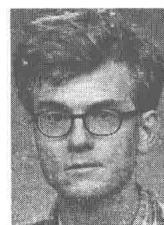
Troy Baer 任职于田纳西大学和橡树岭国家实验室，领导美国计算科学国家研究所 (NICS) 系统和运营组的 HPC 系统团队。他一直参与大型系统部署，包括 Beacon、Nautilus 和 Kraken。2014 年 4 月，Troy 获得自适应计算终身成就奖，以表彰其使用 Moab 对调度和资源管理所做出的贡献。



Cédric Andreolli 是法国 Intel 公司能源组的一名应用工程师。他帮助石油和天然气行业优化运行在 Intel 平台上的应用程序。



Carsten Benthin 是德国 Intel 公司的一名图像研究科学家。他的研究方向包括射线追踪和高性能渲染、吞吐量和高性能计算、底层代码优化以及大规模并行硬件体系结构。



Edoardo Aprà 是美国太平洋西北国家实验室 (PNNL) 环境分子科学实验室的首席科学家。他的研究方向是高性能计算算法和软件开发 (尤其是化学应用)。他是 NWChem 包中分子密度函数理论 (DFT) 的主要开发人员。



Per Berg 任职于丹麦气象研究所。他把数学模型和科学计算教育应用于水环境 (河口、海洋) 中的应用程序开发建模软件。同时为私企和国企工作，Per 参与了众多应用模型解决工程和科学问题的项目。



Vincent Betro 任职于美国计算科学国家研究所，田纳西大学和橡树岭国家实验室。他的研究集中于为多个架构移植和优化应用程序，尤其是 Intel Xeon Phi，并开发计算流体动力学代码。他也是 XSEDE 项目的培训经理，并且在此期间他一直侧重于为 Stampede 和 Beacon 开发 Xeon Phi 协处理器培训教材。



Leonardo Borges 是美国 Intel 公司的一名资深工程师，并且从初期就开始参与 Intel 集成众核架构编程。他专门研究 HPC，并将其应用于数值分析和开发并行数值数学库。Leo 专注于同石油和天然气行业相关的优化工作。



Ryan Braby 任职于田纳西大学和橡树岭国家实验室，是美国计算科学联合研究所（JICS）的网络基础设施首席官。Ryan 一直直接参与管理和部署在 Top 500 列表中排名第一的两个系统、在 Green 500 列表中排名第一的一个系统、在 Top 500 列表中排名前 50 的 18 个系统。



Glenn Brook 任职于田纳西大学和橡树岭国家实验室，目前指导卓越应用加速中心（Application Acceleration Center of Excellence, AACE）并且是 JICS 的首席技术官。他是 Beacon 项目的主要研究者，这个项目由 NSF 和 UT 资助，以研究新型计算技术产生的影响，例如 Intel Xeon Phi 协处理器对计算科学和工程的影响。



Ilya Burylov 是俄罗斯 Intel 公司 Numerics 团队的一名高级软件工程师。他关注统计学、金融和超越数学函数算法的计算优化。Ilya 专注于计算密集分析算法优化和分布式系统中大数据工作流的数据操作步骤。



Ki Sing Chan 是中国香港中文大学的一名本科生，主修数学和信息工程，兼修计算机科学。2013 年暑假期间他第一次在田纳西橡树岭国家实验室参与科研工作。他的研究方向包括大型稠密矩阵的 Cholesky 分解算法的实现。



Gilles Civario 是爱尔兰高阶计算中心（ICHEC）的一名高级软件架构师，致力于为国家服务用户和 ICHEC 的技术转移客户公司设计和实现软硬件解决方案。



Guillaume Colin de Verdière 是法国原子能和替代能源署（CEA）的一名资深专家。他目前的主要研究方向是新型架构，尤其是 Intel Xeon Phi——一种非常有前景的技术，这种技术可能让机器运行速率达到百亿亿级。凭借这一直接研究结果，他正积极研究这种新型技术对于老程序演变的影响。



Eduardo D'Azevedo 是美国橡树岭国家实验室计算数学组的一名研究员 (Staff Scientist)，他的研究兴趣包括开发高度可扩展的并行解算器。他为材料科学领域的项目做出贡献，并致力于通过先进计算 (SciDAC) 程序融入科学发现。他开发了 ScaLAPACK 库的核外和压缩存储扩展，并对最优网格生成做出巨大贡献。



Jim Dempsey 是一名专门从事高性能计算和嵌入式系统优化的咨询师。Jim 是 QuickThread 编程公司的总裁。他的研究方向包括高效编程以及 Intel Xeon 和 Intel Xeon Phi 处理器的优化。



Alejandro Duran 是西班牙 Intel 公司的应用工程师，帮助客户优化代码。自 2005 年以来，他一直是 OpenMP 语言委员会的成员。



Manfred Ernst 是谷歌 Chromium 团队成员。加入谷歌前，他是 Intel 实验室的研究科学家，在那里，他开发了 Embree 光线追踪内核。他的主要研究方向包括真实感渲染、光线追踪的加速结构、采样以及数据压缩。



Kerry Evans 是美国 Intel 公司软件工程师，主要工作是与客户合作优化基于 Intel Xeon 处理器和 Intel Xeon Phi 协处理器的医学影像软件。



Rob Farber 是拥有深厚的 HPC 背景且长期与美国国家实验室和企业合作进行 HPC 优化工作的顾问。Rob 已经撰写并编辑了多本与 GPU 编程相关的书籍，是 TechEnablement.com 的首席执行官和出版商。



Louis Feng 是美国 Intel 公司软件工程师，主要工作是与梦工厂动画公司合作处理高性能图形。他目前的研究兴趣包括光线追踪、基于高度并行架构的真实感图像合成和并行编程模型。



Evgeny Fiksman 任职于美国 Intel 公司，从事基于 x86 平台的视频增强算法优化工作。他是负责 OpenCL 运行时在 Intel CPU 和 Intel Xeon Phi 协处理器上实现的首席架构师和工程师。作为一名面向客户的应用工程师，他目前关注的领域是对金融应用的优化。



Jeff Hammond 是 Intel 实验室并行计算实验室的研究科学家。他的研究方向包括单边和全局视图的编程模型、不规则算法的负载均衡，以及共享内存与分布式内存张量收缩。



Michael Hebenstreit 是美国 Intel 公司资深集群架构师和 Endeavor HPC 基准测试数据中心的技术主管。他帮助这个数据中心成为主要的 HPC 基准测试资源。他创建了第一个 Intel Xeon Phi 协处理器集群，为将 Intel Xeon Phi 协处理器集成到 Endeavour 奠定了基础。



Christopher Hughes 是 Intel 实验室的研究员。他的研究方向是新兴的工作负载和计算机体系结构。目前，他在帮助开发针对计算和数据密集型应用的下一代微处理器，并且专注于宽 SIMD 执行和众核处理器的存储系统。



Sverre Jarp 是挪威 CREN 荣誉成员，曾在 CREN 的 IT 部门工作了 40 多年，推广了先进的、成本效益高的大规模计算。2002 年，他被任命为 CREN 开放实验室的首席技术官，并一直担任这个职位到 2014 年退休。现在，作为 CREN 荣誉成员，他仍致力于研究高吞吐量计算以及基于向量和并行编程的应用可扩展性。



Jim Jeffers 是美国 Intel 公司技术计算组的首席工程师和工程经理。Jim 与人合著了《Intel Xeon Phi Coprocessor High Performance Programming》(Morgan Kaufmann, 2013) 一书。目前，Jim 领导着 Intel 的技术计算可视化团队。



Gregory S. Johnson 是美国 Intel 公司计算机图形学研究员。他的研究领域包括实时和真实感渲染、可见性算法、空间数据结构和图形硬件架构。



Vadim Karpusenko 是美国 Colfax International 公司的首席 HPC 研究工程师。他的研究兴趣包括 HPC 集群的物理建模、高度并行架构和代码优化。



Michael Klemm 是德国 Intel 公司软件与服务团队的高级应用工程师。他主要研究高性能和高吞吐量计算。Michael 是 OpenMP 语言委员会的 Intel 代表，负责 OpenMP 错误处理功能的开发工作。



Karol Kowalski 是美国太平洋西北国家实验室 (PNNL) 环境分子科学实验室的首席科学家。他主要研究精确电子结构方法及其高度可扩展实现。他的方法已用于描述各种多体系统, 包括从原子核和分子, 到分子和纳米科学之间的交叉系统。



Simon McIntosh-Smith 领导英国布里斯托尔大学的一个 HPC 研究小组。他是 ClearSpeed 的联合创始人之一并且是加速 BLAS/LAPACK 和 FFT 库的先驱者。他的研究聚焦在众核算法、性能的可移植性和达到百亿亿级的容错软件技术上。他现在是 Khronos OpenCL 异构编程标准的贡献者。



Michael Lysaght 在爱尔兰高阶计算中心 (ICHEC) 领导新技术活动与英特尔并行计算中心。他专注于支持爱尔兰科学用户社区以及爱尔兰新兴多核和众核技术开发行业。他负责 WP7 “HPC 工具和技术开发” 活动, 这个活动是欧盟的 PRACE 3IP 项目的一部分。



Larry Meadows 是美国 Intel 公司首席工程师, 从 1982 年开始他就为 HPC 开发编译器、相关工具和应用软件。他是 Portland Group 的发起者之一, 从 2004 年他就开始为 Intel 公司工作了。



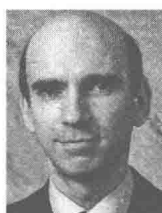
Anton Malakhov 是俄罗斯 Intel 公司的资深软件开发工程师, 研究 Intel 线程构建模块 (TBB)。他优化了 Intel Xeon Phi 协处理器的 TBB 任务调度器, 并发明了一些调度算法, 提高了 Intel OpenCL 运行时在 MIC 架构上的性能。他目前负责 task_arena 和 affinity_partitioner 等 TBB 组件的产品化。



Karl Meerbergen 是比利时鲁汶大学计算机科学系教授。他的研究专注于大规模数值线性代数。



Tim Mattson 是美国 Intel 公司并行计算实验室的首席工程师。他研究大数据应用的软件平台和百亿亿次计算机的执行模型。他拥有超过 30 年的并行计算经验, 并帮助创建了 OpenMP 和 OpenCL。他的著作包括《Patterns for Parallel Programming》《An Introduction to Concurrency in Programming Languages》《OpenCL Programming Guide》。



Iosif Meyerov 是俄罗斯罗巴切夫斯基州立大学 (UNN) 软件系的副主任, 在许多研发项目中担任首席调研员。他的研究兴趣包括高性能计算、科学计算、性能分析和优化、系统编程以及应用数学。



Kent Milfeld 任职于美国德州高级计算中心 (Texas Advanced Computing Center, TACC)。在 UT 的早期他就是高性能计算方面的导师、科学家和 HPC 程序员。在 TACC 用户社区, 他的专家培训为将编程范式映射到高效硬件提供了一种方法, 这样可以获得尽可能高的性能。



Karthik Raman 是美国 Intel 公司的软件架构师, 他专注于性能分析和为 Intel Xeon Phi 优化 HPC 工作负载。他专注于分析优化编译器的代码产生、向量化和评估性能的架构关键特性。他帮助实现变革的方法和工具来提供新的机遇和见解。



Paul Peltz, Jr. 是美国田纳西大学和橡树岭国家实验室国家计算科学研究所的 HPC 系统管理者。他是 Beacon Cluster 的主要管理者并且是 SC NICS 学生集群挑战团队的操作导师。



James Reinders 任职于美国 Intel 公司, 致力于促进在工业中使用并行编程。他的项目包括世界上第一个 TFLOPS 的超级计算机 (ASCI Red) 和世界上第一个 TFLOPS 的微处理器 (Intel Xeon Phi 协处理器)。James 与人合著了《Intel Xeon Phi Coprocessor High Performance Programming》以及其他一些高性能编程书籍。



Simon John Pennycook 是英国 Intel 公司应用工程师, 专注于让开发者充分利用新一代 Intel Xeon Phi 协处理器。他之前主要研究针对大量不同的微处理器架构和硬件平台优化科学应用, 也关注性能的可移植性。



Alexander Reinefeld 是德国 Zuse Institute Berlin (ZIB) 计算机科学系主任和柏林 Humboldt 大学教授。他的研究兴趣包括分布式计算、高性能计算机体系结构、可伸缩和可靠的计算以及点对点算法。



Jacob Weismann Poulsen 是丹麦气象局 (DMI) 研究小组的一个 HPC 和科学编程咨询师, 他擅长分析和优化气象学领域的应用。



Dirk Roose 是比利时鲁汶大学计算机科学系教授。他的研究侧重于计算科学与工程数值方法以及并行科学计算的算法。



Carlos Rosales-Fernandez 是美国德州高级计算中心 (TACC) 高级计算评估实验室的主任, 主要负责新计算机体系结构与高性能计算相关的评估。他是开源 mplabs 代码的作者。



Mikhail Smelyanskiy 是美国 Intel 公司并行计算实验室的一名首席工程师。他负责在当前和未来一代并行处理器系统上设计、实现和分析并行算法和工作负载。他的研究兴趣包括医学成像、计算金融和基本高性能计算内核。



Karl Schulz 在美国 Intel 公司技术计算小组领导 Cluster-Maker 团队研究未来一代 HPC 软件产品。



Thomas Steinke 是德国 ZIB 超级计算机算法和咨询小组的负责人。他的研究兴趣是高性能计算、科学和数据分析应用程序的异构系统以及并行仿真方法。Thomas 在 2004 年参与起草了 OpenFPGA 倡议, 他领导 ZIB 的 Intel 并行计算中心 (IPCC)。



Jason Sewall 是美国 Intel 公司数据中心小组的一名研究人员, 专注于 Intel Xeon Phi 的研究。他的兴趣包括图形、基于物理的建模、并行和高性能计算、数据库、计算金融和图算法。



Shi-Quan Su 任职于美国田纳西大学, 是一个 HPC 顾问, 帮助用户将代码迁移到新的平台。他的主要工作包括针对高转变温度超导材料的大规模粒子的蒙特卡罗模拟。



Gregg Skinner 任职于美国 Intel 公司, 擅长在并行计算机上移植和优化科学与工程应用。他于 2011 年加入 Intel 公司软件和服务部门。



Alexander Sysoyev 是俄罗斯罗巴切夫斯基州立大学 (UNN) 软件系的副教授, 在许多研发项目中担任首席调研员。他的研究兴趣包括高性能计算、全局优化、性能分析和优化、系统编程以及应用数学。



Philippe Thierry 任职于法国 Intel 公司，带领能源工程团队为能源领域的用户和软件厂商提供支持。他的工作包括为当前和未来平台以及与应用程序行为有关的

超级计算机定义分析与调整 HPC 应用程序。他的研究面向百亿亿次计算的性能推断、应用特征和建模。



Ingo Wald 是美国 Intel 公司的一名研究科学家。他的研究方向涉及光线追踪和照明仿真、实时图形、并行计算以及常规视觉 / 高性能计算。



Antonio Valles 是美国 Intel 公司高级软件工程师，负责 Intel Xeon Phi 协处理器的性能分析与优化。他在 Intel 分析、优化的软件涵盖客户端、移动业务和 HPC 业务。

他非常喜欢写代码并编写了内部的 post-Si 和 pre-Si 工具以帮助分析和优化应用程序。



Florian Wende 是德国 ZIB 分布式算法和超算部门可扩展算法研究小组的一员。他对计算机科学和计算物理中应用的加速器和众核计算非常感兴趣。他的研究

方向是不规则并行计算的负载均衡和接近硬件级的代码优化。



Jerome Vienne 是美国德州高级计算中心 (TACC) 在德州大学奥斯汀分校的研究助理。他的研究兴趣包括性能分析和建模、高性能计算、高性能网络、基准测试

和百亿亿次计算。



Kwai Lam Wong 是美国田纳西大学计算机科学联合研究院 (JICS) 的校园项目组副主任。他是田纳西大学诺克斯维尔分校机械、航空和生物学工程的 CFD

实验室主任。他的研究兴趣包括数值线性代数、并行计算、计算流体动力学和有限元法。



Andrey Vladimirov 是美国 Colfax International 公司 HPC 研究小组的负责人。他的研究内容主要集中在把现代计算技术应用于需要大量计算的科学问题。



Sven Woop 是德国 Intel 公司的一名图形研究科学家，他在这里开发了 Embree 光线追踪内核。他的研究兴趣包括计算机图形学、硬件设计和并行编程。



Claude Wright 是美国 Intel 公司从事 Intel Xeon Phi 协处理器功耗分析和 Green 500 系统调优的工程师。



Albert-Jan Nicholas Yzelman 是比利时鲁汶大学计算机科学学院的一名博士后研究员。他在 ExaScience 生命实验室从事稀疏矩阵计算、高性能计算和通用并行编程方面的研究。



Rio Yokota 是沙特阿拉伯阿卜杜拉国王科技大学 (KAUST) 极限计算研究中心的一名研究科学家。他是 FMM 库 ExaFMM 的主要开发者。他所在团队于 2009 年在 760 GPU 上凭借 FMM 代码在价格 / 性能方面获得了戈登贝尔奖。现在他主要在 Titan、Mira、Stampede、K computer 和 TSUBAME 2.5 等架构上优化 ExaFMM 代码。



Weiqun Zhang 是美国劳伦斯伯克利国家实验室计算科学与工程中心的一员。他的研究兴趣是高性能计算、偏微分方程的数值计算方法、燃烧和天体物理学等科学和工程领域的应用。



Charles Yount 是美国 Intel 公司软件和服务部门的首席工程师，自 1995 年开始一直在这里工作。他的研究方向是计算机性能分析和优化 (pre-Si 和 post-Si)、面向对象的软件设计、机器学习和计算机体系结构。

出版者的话

译者序

推荐序

前 言

作者简介

第 1 章 引言 1

1.1 学习成功经验 1

1.2 代码现代化 1

1.3 并发算法现代化 1

1.4 向量化和数据局部性现代化 2

1.5 理解功耗使用 2

1.6 ISPC 和 OpenCL 2

1.7 Intel Xeon Phi 协处理器特性 2

1.8 众核和新异构系统 2

1.9 书名中没有 Xeon Phi 与新异构
架构编程 3

1.10 众核的未来 3

1.11 下载 3

1.12 更多信息 4

第 2 章 从正确到正确 & 高效: Godunov 格式的 Hydro2D 案例学习 5

2.1 现代计算机上的科学计算 5

2.1.1 现代计算环境 6

2.1.2 CEA 的 Hydro2D 6

2.2 冲击流体动力学的一种数值方法 7

2.2.1 欧拉方程 7

2.2.2 Godunov 方法 7

2.2.3 哪里需要优化 9

2.3 现代计算机架构的特征 9

2.3.1 面向性能的架构 9

2.3.2 编程工具和运行时 10

2.3.3 计算环境 11

2.4 通向高性能的路 11

2.4.1 运行 Hydro2D 11

2.4.2 Hydro2D 的结构 12

2.4.3 优化 15

2.4.4 内存使用 16

2.4.5 线程级并行 17

2.4.6 算术效率和指令级并行 24

2.4.7 数据级并行 26

2.5 总结 32

2.5.1 协处理器与处理器 32

2.5.2 水涨船高 32

2.5.3 性能策略 33

2.6 更多信息 34

第 3 章 HBM 上的 SIMD 与并发 优化 36

3.1 应用程序: HIROMB-BOOS-
MODEL 36

3.2 关键应用: DMI 36

3.3 HBM 执行配置文件 37

3.4 HBM 优化综述 38

3.5 数据结构: 准确定位位置 38

3.6 HBM 上的线程并行 41

3.7 数据并行: SIMD 向量化 45

3.7.1 零散的可优化部分 46

3.7.2 过早抽象是万恶之源 48

3.8 结果 50

3.9 详情分析 51

3.10 处理器与协处理器可扩展性
对比 52

3.11 CONTIGUOUS 属性 53

3.12 总结 54

3.13 参考文献 54

3.14 更多信息 55

第 4 章 流体动力学方程优化	56	6.1.3 程序实例中的故障树：基本模拟	93
4.1 开始	56	6.2 实例实现	94
4.2 1.0 版本：基础版本	57	6.3 其他因素	101
4.3 2.0 版本：线程盒	59	6.4 总结	101
4.4 3.0 版本：栈内存	63	6.5 更多信息	101
4.5 4.0 版本：分块	63	第 7 章 深度学习的数值优化	102
4.6 5.0 版本：向量化	64	7.1 拟合目标函数	102
4.7 Intel Xeon Phi 协处理器上的运行结果	68	7.2 目标函数与主成分分析	105
4.8 总结	69	7.3 软件及样例数据	106
4.9 更多信息	70	7.4 训练数据	109
第 5 章 分阶段准同步栅栏	71	7.5 运行时间	109
5.1 如何改善代码	74	7.6 扩展结果	111
5.2 如何进一步改善代码	74	7.7 总结	111
5.3 超线程方阵	74	7.8 更多信息	112
5.4 关于该方案哪些地方不是最优的	75	第 8 章 优化聚集 / 分散模式	113
5.5 超线程方阵编码	76	8.1 聚集 / 分散在 Intel 架构下的说明	114
5.5.1 如何确定内核间兄弟线程和内核内 HT 线程	77	8.2 聚集 / 分散模式在分子动力学中的应用	115
5.5.2 超线程方阵手动分区方法	77	8.3 优化聚集 / 分散模式	117
5.5.3 吸取教训	79	8.3.1 提高时间和空间的局部性	117
5.6 回到工作	80	8.3.2 选择一种适当的数据布局：AoS 与 SoA	118
5.7 数据对齐	81	8.3.3 AoS 和 SoA 之间的动态转换	119
5.7.1 尽可能使用对齐的数据	81	8.3.4 分摊聚集 / 分散和转换的开销	122
5.7.2 冗余未必是件坏事	81	8.4 总结	123
5.8 深入讨论分阶段准同步栅栏	84	8.5 更多信息	123
5.9 如何节省时间	86	第 9 章 N 体问题直接法的众核实现	125
5.10 几个留给读者的优化思考	90	9.1 N 体模拟	125
5.11 类似 Xeon Phi 协处理器的 Xeon 主机性能优化	91	9.2 初始解决方案	125
5.12 总结	92	9.3 理论极限	126
5.13 更多信息	92	9.4 降低开销和对齐数据	128
第 6 章 故障树表达式并行求解	93		
6.1 动机和背景	93		
6.1.1 表达式	93		
6.1.2 表达式选择：故障树	93		

9.5 优化存储层次	131	12.5 更多信息	174
9.6 改进分块	133	第 13 章 MPI 和异构计算	175
9.7 主机端的优化	135	13.1 现代集群中的 MPI	175
9.8 总结	136	13.2 MPI 任务地点	176
9.9 更多信息	136	13.3 DAPL 提供者的选择	180
第 10 章 N 体方法	137	13.3.1 第一个提供者 OFA-V2- MLX4_0-1U	180
10.1 快速 N 体方法和直接 N 体 内核	137	13.3.2 第二个提供者 ofa-v2-scif0 以及 对节点内部结构的影响	180
10.2 N 体方法的应用	138	13.3.3 最后一个提供者	181
10.3 直接 N 体代码	138	13.3.4 混合程序的可扩展性	182
10.4 性能结果	141	13.3.5 负载均衡	184
10.5 总结	142	13.3.6 任务和线程映射	184
10.6 更多信息	142	13.4 总结	185
第 11 章 使用 OpenMP 4.0 实现 动态负载均衡	144	13.5 致谢	185
11.1 最大化硬件利用率	144	13.6 更多信息	185
11.2 N 体内核	146	第 14 章 Intel Xeon Phi 协处理器 功耗分析	186
11.3 卸载版本	149	14.1 功耗分析	186
11.4 第一个处理器与协处理器 协作版本	150	14.2 用软件测量功耗和温度	187
11.5 多协处理器版本	152	14.2.1 创建功耗和温度监控脚本	188
11.6 更多信息	155	14.2.2 使用 micsmc 工具创建功耗和 温度记录器	189
第 12 章 并发内核卸载	156	14.2.3 使用 IPMI 进行功耗分析	190
12.1 设定上下文	156	14.3 基于硬件的功耗分析方法	192
12.1.1 粒子动力学	156	14.4 总结	196
12.1.2 本章结构	157	14.5 更多信息	196
12.2 协处理器上的并发内核	158	第 15 章 集成 Intel Xeon Phi 协 处理器至集群环境	197
12.2.1 协处理器设备划分和线程 关联	158	15.1 早期探索	197
12.2.2 并发数据传输	163	15.2 Beacon 系统的历史	197
12.3 在 PD 中使用并发内核卸载进行 作用力计算	166	15.3 Beacon 系统的架构	198
12.3.1 使用牛顿第三定律并行评估 作用力	166	15.3.1 硬件环境	198
12.3.2 实现作用力并发计算	167	15.3.2 软件环境	198
12.3.3 性能评估：之前与之后	171	15.4 Intel MPSS 安装步骤	199
12.4 总结	173	15.4.1 系统准备	199
		15.4.2 安装 Intel MPSS 栈	200

15.4.3 生成和定制配置文件	201	17.3.1 全局数组	225
15.4.4 MPSS 升级	204	17.3.2 张量收缩引擎	226
15.5 建立资源和工作负载管理器	204	17.4 设计卸载解决方案	226
15.5.1 TORQUE	204	17.5 卸载架构	229
15.5.2 序言程序	205	17.6 内核优化	230
15.5.3 尾声程序	206	17.7 性能评估	232
15.5.4 TORQUE/ 协处理器集成	207	17.8 总结	233
15.5.5 Moab	207	17.9 致谢	235
15.5.6 提高网络局部性	207	17.10 更多信息	235
15.5.7 Moab/ 协处理器集成	207		
15.6 健康检查和监控	208	第 18 章 大规模多系统上的高效	
15.7 常用命令脚本化	209	嵌套并行	238
15.8 用户软件环境	210	18.1 动机	238
15.9 今后的方向	211	18.2 基准测试	238
15.10 总结	212	18.3 基线基准测试	239
15.11 致谢	212	18.4 流水线方法——Flat_arena 类	240
15.12 更多信息	212	18.5 Intel TBB 用户管理任务调度	
		平台	241
第 16 章 在 Intel Xeon Phi 协处理器		18.6 分层方法——Hierarchical_	
上支持集群文件系统	214	arena 类	243
16.1 网络配置概念和目标	214	18.7 性能评估	243
16.1.1 网络选项概览	215	18.8 对 NUMA 架构的影响	245
16.1.2 设置集群启用协处理器的		18.9 总结	246
步骤	216	18.10 更多信息	246
16.2 协处理器文件系统支持	217		
16.2.1 支持 NFS	217	第 19 章 Black-Scholes 定价的性能	
16.2.2 支持 Lustre 文件系统	218	优化	248
16.2.3 支持 Fraunhofer BeeGFS 文件		19.1 金融市场模型基础及 Black-Scholes	
系统	219	公式	248
16.2.4 支持 Panasas PanFS 文件		19.1.1 金融市场数学模型	248
系统	220	19.1.2 欧式期权和公平价格概念	249
16.2.5 集群文件系统的选择	220	19.1.3 Black-Scholes 公式	250
16.3 总结	220	19.1.4 期权定价	250
16.4 更多信息	221	19.1.5 测试平台架构	250
		19.2 案例研究	251
第 17 章 NWChem: 大规模量子		19.2.1 初始版本——检验正确性	251
化学仿真	222	19.2.2 参照版本——选择合适的	
17.1 引言	222	数据结构	251
17.2 回顾单线程 CC 形式	222	19.2.3 参照版本——不要混合使用	
17.3 NWChem 软件架构	225	数据类型	252

19.2.4	循环向量化	253	22.4	OpenCL 与 Intel Xeon Phi 协处理器	285
19.2.5	使用快速数学函数: erff() 与 cdfnormf()	255	22.5	性能评估	285
19.2.6	代码等价变换	256	22.6	案例研究: 分子对接算法	287
19.2.7	数组对齐	257	22.7	性能评估: 性能可移植性	289
19.2.8	尽可能降低精度	258	22.8	相关工作	291
19.2.9	并行工作	259	22.9	总结	291
19.2.10	使用热身	260	22.10	更多信息	291
19.2.11	使用 Intel Xeon Phi 协处理器实现轻松移植	261	第 23 章 应用到 Stencil 计算中的特性提取和优化方法		
19.2.12	使用 Intel Xeon Phi 协处理器实现并行工作	261	23.1	引言	292
19.2.13	使用 Intel Xeon Phi 协处理器和流存储	262	23.2	性能评估	293
19.3	总结	263	23.2.1	测试平台的 AI	293
19.4	更多信息	264	23.2.2	内核的 AI	294
第 20 章 使用 Intel COI 库传输数据			23.3	标准优化	296
20.1	使用 Intel COI 库的第一步	265	23.3.1	自动应用调试	301
20.2	COI 缓冲区种类和传输性能	266	23.3.2	自动调试工具	304
20.3	应用程序	269	23.3.3	结果	305
20.4	总结	270	23.4	总结	305
20.5	更多信息	270	23.5	更多信息	307
第 21 章 高性能光线追踪			第 24 章 剖析指导优化		
21.1	背景	272	24.1	计算机科学中的矩阵转置	308
21.2	向量化的光线遍历	272	24.2	工具和方法	309
21.3	Embree 光线追踪内核	273	24.3	串行: 初始的就地转置实现	310
21.4	在应用程序中使用 Embree	274	24.4	并行: 使用 OpenMP 增加并行度	313
21.5	性能	276	24.5	分块: 提高数据局部性	315
21.6	总结	277	24.6	规范化: 多版本微内核	319
21.7	更多信息	278	24.7	预组织: 释放更多的并行性	322
第 22 章 OpenCL 程序的可移植性能			24.8	总结	326
22.1	两难的困境	279	24.9	更多信息	327
22.2	OpenCL 简介	280	第 25 章 基于 ITAC 的异构 MPI 应用优化		
22.3	OpenCL 示例: 矩阵乘	282	25.1	亚式期权定价	328
			25.2	应用设计	329
			25.3	异构集群中的同步	330

25.4	通过 ITAC 寻找性能瓶颈	331	27.2	稀疏矩阵数据结构	353
25.5	建立 ITAC	331	27.2.1	压缩后的数据结构	354
25.6	非均衡的 MPI 运行	332	27.2.2	分块	356
25.7	手动负载均衡	335	27.3	并行 SpMV 乘法	356
25.8	动态老板 - 工人负载均衡	337	27.3.1	部分分布式并行 SpMV	356
25.9	结论	339	27.3.2	完全分布式并行 SpMV	357
25.10	更多信息	340	27.4	Intel Xeon Phi 协处理器的 向量化	358
第 26 章 集群上可扩展 OOC 解法器			27.5	评估	362
26.1	引言	341	27.5.1	Intel Xeon Phi 协处理器	363
26.2	基于 ScaLAPACK 的 OOC 分解 算法	342	27.5.2	Intel Xeon 处理器	365
26.2.1	核内分解	342	27.5.3	性能比较	366
26.2.2	OOC 分解	343	27.6	总结	366
26.3	从 NVIDIA GPU 移植到 Intel Xeon Phi 协处理器	344	27.7	致谢	367
26.4	数值结果	346	27.8	更多信息	367
26.5	结论和展望	350	第 28 章 基于 Morton 排序的性能 优化		
26.6	致谢	350	28.1	通过数据重排提高缓存局 部性	368
26.7	更多信息	350	28.2	性能改进	368
第 27 章 稀疏矩阵向量乘：并行化和 向量化			28.3	矩阵转置	369
27.1	引言	352	28.4	矩阵乘法	373
			28.5	总结	377
			28.6	更多信息	378

引言

James Reinders

Intel 公司

《Intel Xeon Phi Coprocessor High-Performance Programming》出版后，我和 Jim Jeffers 最常听到的建议就是我们需要写本“参考大全”。Guillaume Colin de Verdière 是早期推荐我们写这本书的人之一，他为我们能够开展这个项目而感到高兴。Guillaume 也通过实际行动来帮助我们推进这个项目。他与 Jason Sewall 合著了第 2 章。这一章节反映了这本书的基本前提，即成功经验的分享对他人是很有教育意义的。它还包含了一个在 Intel Xeon Phi 系列上进行大规模并行编程的人很熟悉的主题：在 Intel Xeon Phi 协处理器上运行代码是非常容易的。这可以使你快速地投入优化和高性能的实现工作中，同时，我们也确实需要调整应用程序的并行部分！值得注意的是，我们发现这样的优化工作能够同时提高处理器和协处理器的性能。正所谓“水涨船高”。

1.1 学习成功经验

本书的主要内容就是向他人学习。本书汇集了许多并行编程领域专家的工作成果。其中的例子是根据教育意义、适用性和所取得的成果来进行筛选的。并且，可以下载代码，然后自己尝试运行。所有这些例子都使用并行编程领域非常成功的优化方法，但是并不是所有例子都扩展得足够好，使其在 Intel Xeon Phi 协处理器上的性能比在处理器上好。实际上，我们需要面对并指出的是：通用的编程模型非常重要。我们常常可以看到这个想法一次又一次在实际例子中出现，也包括在本书的实例中出现。

我们非常感谢为本书做出贡献的人们。在这本书中，你可以发现丰富的案例和建议。因为这是引言，所以我们将对这些案例和建议进行总结。希望你可以认真地从第 2 章开始，学习本书中的案例。

1.2 代码现代化

最近，“代码现代化”这个词非常火。受 61 个内核所鼓舞，越来越多的人开始讨论它。在本书中就有很多地方体现着“现代化”。

代码现代化就是将代码进行重组，或许还需要改写算法来提高线程并行性、向量 / SIMD 操作和计算强度，以实现在现代架构上的性能优化。线程并行性、向量 / SIMD 操作和强调时态数据复用，对高性能编程均非常重要。许多已有的应用程序是在这些性能优化方法出现之前完成的，因此这样的代码还没有针对现代计算机进行优化。

1.3 并发算法现代化

重新考虑算法使其更加适应现代计算机并行性的案例遍布全书。第 5 章鼓励使用栅栏以

增加程序并发度。第 11 章强调非静态工作负载分解的重要性，因为工作负载和设备都不会真正均衡地运行。第 18 章展示并行世界是非平坦的观念的重要性。第 26 章通过调整数据、计算和存储来提升性能。第 12 章通过在异构节点上确保并行性来提升性能。第 13 章、第 25 章说明如何在异构集群上增强并行性。

1.4 向量化和数据局部性现代化

第 8 章详细讨论数据布局问题，用于处理数据向量化。第 27 章和第 28 章进一步说明了对数据进行布局和向量化的重要性。

1.5 理解功耗使用

功耗使用情况在很多章节都有提及。我们邀请了 Intel 功耗调优专家 Claude Wright 来写第 14 章。这一章着重描述功耗测量方法，包括通过 Intel MPSS 工具创建一个基于软件的简单功耗分析器。并且，该章也阐述了测量空闲功耗的困难性，因为当程序忙于测量功耗时，就已经不是空闲状态了。

1.6 ISPC 和 OpenCL

OpenMP 和 TBB 在工业界并行编程解决方案中处于领导地位。除了介绍这两个方法以外，本书还引入了其他解决方案的案例。

SPMD 编程提供了向量化的有趣解决方案，包括辅助数据布局，但是这个方法会影响数据的顺序一致性。这样可行吗？第 6 章和第 21 章使用 ISPC 和 SPMD 方法作为判断的依据。即使在不使用 ISPC 的情况下，SPMD 也会符合你想要向量化的想法。

第 22 章用于推进 OpenCL 在异构环境下的使用。该章描述了 BUDE 分子对接代码的测试结果。这段代码可以在多种系统下获得高于 30% 的浮点计算峰值性能。

1.7 Intel Xeon Phi 协处理器特性

本书中大部分章节的内容都是将算法移植到处理器和协处理器上，但其中有 3 章是用于深入分析 Intel Xeon Phi 协处理器特性的。第 15 章提供在集群系统中管理 Intel Xeon Phi 协处理器的最佳实践。第 16 章和第 20 章为 Intel Xeon Phi 协处理器使用者提供了有价值的观点。

1.8 众核和新异构系统

自从 2012 年 11 月 Intel Xeon Phi 协处理器首次发布以来，其使用率一直在稳步提升。到 2013 年年中，Intel Xeon Phi 协处理器在 TOP 500 设备中所贡献的浮点计算量超过了所有作为浮点计算加速器的 GPU 所贡献的计算量。事实上，在 TOP 500 超级计算机中，只有 Intel Xeon 处理器所贡献的浮点计算量比 Intel Xeon Phi 协处理器贡献的多。

就像在前言中所提到的，Intel Xeon Phi 协处理器的 61 个内核已经开启了并行编程的新纪元。如《Intel Xeon Phi Coprocessor High-Performance Programming》所介绍的，这个协处理器采用与处理器相同的编程语言、并行编程模型和工具。这意味着在协处理器上编程的挑战和在通用处理器上是一样的。这是因为在处理器和 Intel Xeon Phi 协处理器的设计中都尽量避免会使编程受限的异构特征。

使用 Intel Xeon Phi 协处理器的程序员的经验一次又一次印证了通用编程模型的重要性。这一点在本书各个章节中也重复强调。要传达给读者的信息是，很明显在 Intel Xeon Phi 协处理器上进行扩展和向量化调优所花费的时间就是处理器（如 Intel Xeon 处理器）性能调优的时间。

1.9 书名中没有 Xeon Phi 与新异构架构编程

因为关键的编程挑战基本都是并行，所以我们强调在多核和众核计算上的适用性，而非仅仅强调在 Intel Xeon Phi 协处理器上的适用性，所以本书的书名中并没有“Xeon Phi”。

但是，处理器和协处理器组成的系统的确在两个主要问题上更显优势，而这两个问题也在本书中得到了解决：1）隐藏在主机和附属设备间的数据移动延时，其中设备主要是指 GPU 和协处理器，未来 Intel Xeon Phi 系列产品将通过作为处理器而非捆绑的协处理器来提供能够消除数据移动开销的配置。2）另一个独特且广泛的挑战是异构系统编程。以前，异构编程是指包含互不兼容的计算设备的系统。不兼容导致需要使用不同的编程模型，甚至不同的开发工具和编码方法。Intel Xeon Phi 产品将这一情况彻底改变。Intel Xeon Phi 协处理器为并行编程提供了与所有处理器兼容的编码方法。Intel 的用户将其命名为“新异构架构”，以强调在探索异构系统的过程中，终于不再需要编程也是异构的。能够在计算设备间尤其是处理器和协处理器间使用通用的编程模型，给了我们在新异构架构硬件上非常令人满意的编程模型。

1.10 众核的未来

Intel 已经宣布开始研究多种未来 Intel Xeon Phi 设备，并且已经发布了关于第二代产品 Knights Landing 的信息。

摩尔定律为我们带来的增长在 Knights Landing 中得到明显体现。最大的突破是将 Knights Landing 作为处理器。能够作为一个处理器，益处是巨大的，包括会有更高的能效，能够减少数据移动，大的标准内存，以及对 Intel AVX-512 的支持（处理器兼容的向量能力与第一代 Intel Xeon Phi 协处理器的向量能力几乎一样）。

摩尔定律为我们带来的其他好处包括使用非常现代化的乱序低功耗内核设计，支持封装内的高带宽内存，以及对集成结构的支持（包括网络接口控制器）。

为 Knights Landing 做准备的最好方法是彻底地被 61 个内核所激发。Intel Xeon Phi 协处理器的性能调优是确保应用程序可以很好地利用 Knight Landing 的资源最好的方式。当然，因为 Knights Landing 预计是更具有通用性的，所以使用超过 50 个内核且基于处理器的系统可能是使你的应用程序为 Knights Landing 做好准备的更好选择。需要超过 50 个内核，是因为根据以往经验少量的内核将无法体现利用高层次并行性的性能调优效果。

逻辑上可以确定，众核的未来是光明的。新异构编程使并行性的代码现代化成为可能。同时，在之后的每一代 Intel Xeon Phi 设备中，并行性的代码现代化过程似乎将会变得更加简单。我们希望本书中的“技巧”可以激励并帮助你将应用程序现代化，以利用高度并行化的计算机。

1.11 下载

在本书的创作过程中，我们特意要求各个章节的作者着重描述能够突出其核心概念的

重要代码片段。另外，我们要求各个章节都有完整的例子，这些例子可以在配套网站 <http://lotsofcores.com> 或项目网站上下载。每章最后一节会告诉你在哪里可以下载代码。

教师以及所有需要做相关报告的人都会体会到能够在 <http://lotsofcores.com> 下载本书中所有数据、图表和照片的价值。请使用这些资源来为更多的软件开发人员教授和解释并行编程！我们非常感谢你能够在描述资源来源时提到我们，否则，我们将为这些资源的使用设置一些条件。

1.12 更多信息

一些值得阅读的额外内容如下：

- 与本书相关的下载网站：<http://lotsofcores.com>。
- 由 Intel 赞助的并行编程网站：<http://go-parallel.com>。
- Intel 网站上关于 Intel Xeon Phi 产品的信息：<http://intel.com/xeonphi>。
- Intel 网站上关于在 Intel Xeon Phi 产品上编程的信息：<http://intel.com/software/mic>。
- Intel Xeon Phi 用户组网站：<https://www.ixpug.org>。
- 高级向量指令的信息：<http://software.intel.com/avx>。

从正确到正确 & 高效：Godunov 格式的 Hydro2D 案例学习

Jason D. Sewall^{*}, Guillaume Colin de Verdière[†]

^{*} 美国, Intel 公司; [†] 法国, CEA

在计算机上高效模拟物理过程是个相当复杂的过程。一方面现实的物理现象非常微妙且复杂, 另一方面现代计算机系统又非常繁杂。同时, 即使是高效模拟非常相似的物理现象, 随着细节的增加和计算环境的变化, 模拟问题的复杂程度也会随着增大, 如模拟海洋中的波浪运动和摩托艇产生的波浪。

性能是现在计算所关注的焦点, 而并行性是这些焦点中的特性之一。因此在这些现代系统上实现物理过程的高效模拟必须考虑这些性能特点。

本章将探讨一段科学模拟代码, 这段代码是一个以气体动力学为基础的模拟程序。这份程序的输出结果正确, 但(初始版本)性能欠佳。通过分析代码的实现并介绍代码映射到现代架构的方式, 我们将看到它的性能将如何显著提升。

2.1 现代计算机上的科学计算

在电子计算机出现前, 数值分析与求知欲一直是计算机发展的动力。从古代的算盘到纳皮尔筹, 再到差分机和分析机, 人们逐渐认识到使用大量的运算几乎可以描述我们周围的一切事物, 这种思想推动了进行快速、可靠、精确计算的技术与设备的发展。

现代世界的科学计算在大型计算机领域的发展中发挥着巨大作用。超级计算机就是为模拟天气、材料试验、探索宇宙而建造的。用于构建超级计算机、笔记本电脑甚至手机的芯片都是通过“处理数据”(该术语源自科学计算领域)的能力来评估自身。

科学计算能够持续地得以进化和发展, 其部分原因是由于人们对规模更大、描述更详细的问题的求知欲。只要有足够的资源, 化学家将非常乐意增加分子动力学模拟中原子的数量, 海洋学家将使用更多的网格单元模拟海洋, 天体物理学家肯定也在设想着使用更多星体的模型。宇宙可能是有限的, 但科学计算量可能是无限大的。

正确性挑战。除计算工作需要大量的计算资源之外, 保证数值计算工作的正确执行是所有数值计算研究人员的另一项挑战。由于数字运算是近似的, 因此在将许多优美的抽象数学结构应用到离散环境时要格外谨慎, 甚至在有些情况下, 离散化应用某些公式是完全不适合的。

在进行科学计算的代码开发中需要足够的耐心并遵循大量规则以保证程序的正确性: 这个数学模型是否能够合理地表述现象? 数值是否稳定? 是否能够正确地选择参数? 代码实现是否正确? 硬件是否可靠?

在某些情况中, 质量较差的代码依然可能得出正确的结果, 但是会出现比较好的代码执

行慢、优化器难处理等情况。

定制的工具。软件开发者深知标准化工具（比如软件库）带来的好处，这些标准化的工具可以正确传递开发者的意图。然而，这些软件库必须定制。虽然 LAPACK/BLAS 是一个非常有用的工具，但是并非所有的代码（比如本章所用的代码）都适合调用这些软件库中的函数。

幸运的是，编译器（可能是最普遍使用的软件工具）技术一直随着硬件技术的发展而不断优化。因此这些优化后的编译器能够帮助程序员在不同架构特征的平台上正确执行某一部分代码，同时也能自动处理许多低层次细节，如应用向量指令以及对线程进行自动整理。

2.1.1 现代计算环境

编译器是我们进行代码开发环境中的一个重要组成部分。主要包括以下几个部分。

- 硬件：处理器的微观 / 宏观架构、I/O、通信接口和集群级别内容。
- 运行时：操作系统和用于管理硬件资源并为用户提供抽象方法的运行时软件。
- 开发工具：编译器、调试器以及帮助程序员将想法转化成正确且有效代码的分析工具。

计算环境的各部分都在代码开发、维护和运行中发挥重要作用。新的计算环境优化在不断发展和使用。

科学家程序员困境。科学代码通常由熟悉理论的领域专家所写。这些科学家 - 程序员非常熟悉不断发展的计算环境，但并非总紧跟着计算环境发展的前沿。一个科学代码库的维护和开发可能超过十年（或 30 年），新方法兼容前期用户几乎是不可能的任务。

考虑到上述这些挑战与得到正确结果的任务，程序员所编写的代码往往不能达到最优的性能。直接快速开发的代码和专家经过数月研究所实现的代码在性能上的巨大差距称为“忍者差距”。该性能差异的量级是计算机开发商、编译器作者和开发人员所热切关注的焦点。这种差距可以反映出处理器性能的广泛实用性和编译器的质量。“忍者差距”存在于所有代码中，但通过本章的学习我们可以在实际应用中缩小这种差距。

2.1.2 CEA 的 Hydro2D

本案例研究的对象是一段由 CEA 所开发的名为 Hydro2D 的中等规模代码。CEA (Commissariat à l'énergie atomique et aux énergies) 是一所法国政府机构，该机构主要进行民用和军用研发。Hydro2D 代码用于内部天体动力学的“代理”程序。这类程序旨在保证代码量（数十万行代码，而不是几十万或几百万行代码）可控与保护原始代码知识产权的情况下，挖掘“基准”应用的基本计算特性。

参考的 Hydro2D 代码大约有 500 行。这份代码主要用于解决“冲击流体动力学”问题。事实上，就是高能气体在二维坐标上的移动问题。除了在天体动力学上的应用之外，该代码使用的技术与航天领域的部分代码十分相似。

原始代码具有短小紧凑、易读性强且执行结果正确的特点。然而，其在多核架构上性能较差。其中的某些性能问题是为了保证代码的可读性和简单性而产生的，但其他问题是由于忽略了面向性能的编程所造成的。由于对运行时和硬件行为的错误理解，参考代码的某些方面性能低下。

章节安排。在接下来的章节中，我们将回顾参考代码的一些背景知识以及要解决的问题，讨论现代计算环境上的重要特征，并且研究如何将这段代码转换成高效代码。

我们将量化每次修改后得到的性能提升。其中我们重点关注多核编程环境下的单节点性能提升，同时检查“起始错误”的后果，也将看到在硬件模型不完全清楚的情况下如何优化。实验结果表明，在不同规模问题的中计算环境的各参数能够得到 3 ~ 10 倍的性能提升。

2.2 冲击流体动力学的一种数值方法

在深入了解如何优化这段代码之前，首先介绍并分析该代码所要解决的问题及其使用的高级算法。

2.2.1 欧拉方程

欧拉方程（或者欧拉系统方程）是一个偏微分方程组，该方程是流体动力学中 Navier-Stokes 方程的特例。其中的流体是无黏性流体，也就是说，流体的黏度影响是可以忽略不计的。二维形式（见本章）为：

$$\frac{\partial}{\partial t} \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E + p)u \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E + p)v \end{bmatrix} = 0 \quad (2-1)$$

其中，

$$E = \frac{p}{\gamma - 1} + \frac{1}{2} \rho (u^2 + v^2) \quad (2-2)$$

这里 ρ 是密度， u 和 v 是 x 和 y 方向上的速度， E 是总能量， p 是压力。注意，这里有 4 个偏微分方程，但是这个系统是欠定的；状态方程（式（2-2））封闭了整个系统并展示了系统中五个变量的依赖关系； γ 表示流体的绝热常数，对空气设置为 1.4。

以守恒形式表示的式（2-1）看上去可能有些奇怪，其中各方程均代表一个守恒量（即质量、动量和能量）。适当离散化之后，求解过程将会保留守恒形式中有用的物理性质。

欧拉系统在很多数学问题中发挥重要作用。它常用于描述气体运动情况并应用在冲击动力学基础研究中。此外，它建立了一个比全 Navier-Stokes 更加简单的系统，且该系统具有重要的非线性行为。从数学方面而言，黏度项的消除将 PDE 的本质变为完全双曲线，这对于缩短求解时间方面的研究具有深远意义。

从式（2-2）上我们可以看出该式所用状态方程为理想气体定律。针对不同的应用已经得到了更加精细的状态方程，但其对该方法性能上的影响较小（访问局部性与这里使用的大同小异）。

2.2.2 Godunov 方法

在冲击的研究中，欧拉方程通常用于对应的计算解决方案中。Hydro2D 是一段冲击捕获代码，使用 Godunov 方法来计算由用户设定的初始边界值问题（IBVP）。

Godunov 方法是在 1959 年由苏联数学家 Godunov 提出的。由于式（2-1）的形式难以在存在冲击时进行定义，因此它使用了 PDE 的积分形式，即各离散单元格对应一个特定量

均值。在每个步长中，我们计算在该步长内发生的各相邻单元格间的平均流量；该流量是指单元格间流动的量。随着执行时间的推移，单元格总数进行更新的同时，解也随之更新。图 2-1 表示的是一维 Godunov 方法示意图。

斜率限制。对于冲击处理而言，采用均值表示法可以起到一定作用，但是若不加以修改，它只能作为一个分段常数方案和并将空间精度限制在一阶上。

目前已经有许多数值计算方法能够在高阶精度的基础上解决该问题。这些方法必须详细规划以确保在求解时的不连续性同时避免引入非物理分散现象。

流量限制是精度改进方法的一种，在加快求解之前，它通过对小流量区域进行研究修改积分方案。Hydro2D 使用另一种名为斜率限制的方法，该方法使用一个邻域的单元格值来计算流量平面中每个单元格的斜率值。

黎曼问题。如何计算流量是 Godunov 方法的核心内容。也就是说，要根据给定的一对共享一条边的单元格，计算出这个时间步长中的平均流量。换言之，即给定两个分段恒定的状态，这两个状态在空间中相遇，那么它们在控制方程的约束下将如何演变？这就是我们熟知的黎曼问题。对于非线性方程组来说，演变过程将会非常复杂。新的中间态可能会冲击左状态、右状态，甚至同时冲击两个状态，更有可能使得两个状态互相冲击。这些新的状态可能包含尖锐、快速移动的脉冲以及缓慢光滑变化的波动块。

我们所需要解决的是一个离散的黎曼问题，这个黎曼问题是在两个共享一条边的单元格间产生的——一个 $n \times n$ 的网格中，每个在每个时间步长中要计算 $2n(n-1)$ 对共享边的单元格对。为了对这些黎曼问题进行求解，研究特定且高效的黎曼求解器一直以来是一个热门的研究内容。

Hydro2D 使用局部非线性求解器来解决黎曼问题：在 2D 欧拉方程中，大量的中间态可以通过在左右状态之间的一个非线性标量方程所决定。我们使用若干次 Newton-Raphson 迭代就会计算出来流量的合适值。

积分。2D 欧拉方程构成了一个双曲系统方程组，这种方程组通常使用显式的技术计算积分。Hydro2D 用显式欧拉方法执行时间的积分。在冲击存在的条件下这是一个简单有效的方法。

另外一个需要考虑的因素是稳定性；对于给定的单元格状态（意味着 Q 和网格间隔 Δx ），为了得到正确稳定的解，有最大可允许的时间步长 Δt 。因为相同的 Δt 必须处处都存在，所以在积分之前 Δt 的值必须已知，而且由于 Δt 依赖于解的离散状态，因此在每个时间步开始计算前都会出现一个计算和归约步的操作。

维度分裂。传统的 Godunov 方法是一维的，对于二维和三维需要使用维度分裂的方法。这是更一般的算子分裂方法的一个特例，在其中，原有方程被算子分裂成几个方程，这些方程就会串行地根据先前的结论（图 2-2）求解。在式（2-1）中有两个空间项被分解成单独的时间步长；为了从 t_n 推进到 t_{n+1} ，解也从使用 $\partial/\partial x$ 项从 t_n 推进到中间态 t_n^* ，接着使用 $\partial/\partial y$ 项推进到 t_{n+1} （流量计算与积分）。

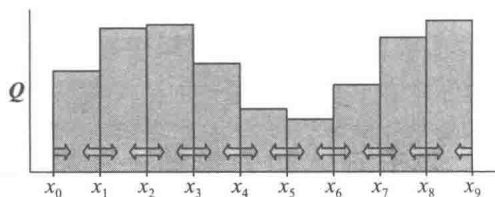


图 2-1 带有未知量 Q 的一维 Godunov 方法， x_i 代表单元的边界，单元格容量代表每个单元格存储的离散均值。箭头代表单元之间的流量交换

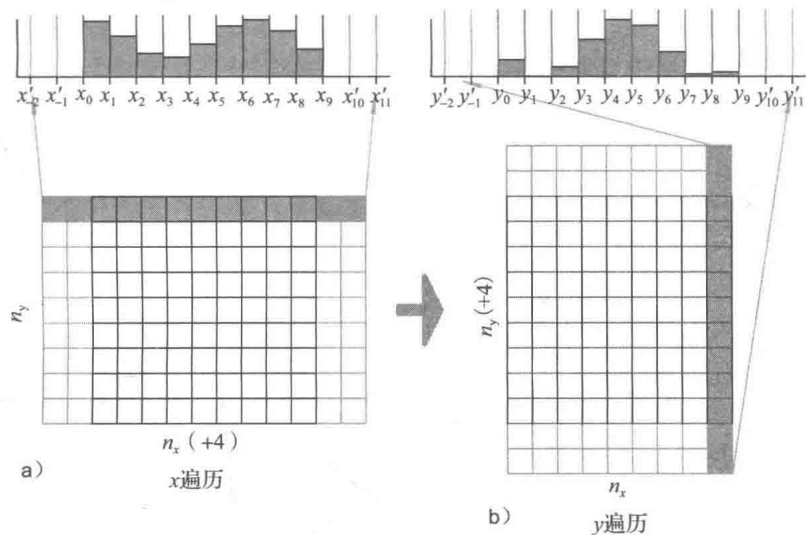


图 2-2 离散维度下的 Godunov 方法

2.2.3 哪里需要优化

PDE 有很多种类并且离散化，积分的数值方法各种各样。对于一个给定的问题，有许多方式可得到这个问题的解。在不同情况下有些方法在特定问题上或者在架构上优于其他方法。

Godunov 方法既不是 stencil 方法（尽管很类似于此），也不能通过对偶线性系统表征，对偶线性系统可借助 Krylov 子空间方法得以求解特征。这意味着关于其他技术的先入为主观念不太可能。

显式 stencil 代码的算术强度通常是非常低的——本质上，它是对每 8 字节（双精度）的数据读后的乘加操作。这使得它们对于快速内存带宽具有要求，除非使用了 blocking/warefront 方法。就像我们将要看到的，这段代码需要计算流量并且对一个 Hydro2D 的单元积分，它对于每个未知项维度有大量的积分操作，需要注意，每个操作都可能成为操作瓶颈。

隐式积分问题通常会导致大规模稀疏矩阵的出现，而且这些大规模稀疏矩阵一定要用迭代方式求解。很自然，这些方法需要很大的工作集；从直觉上，数学操作符的底层“支持”是全局性的。每个未知项都和所有的其他的未知项相耦合，这转化为计算至少需要在每次迭代中使用整个系统矩阵、右边项和未知项。作为一个双曲系统，Hydro2D 中的基础操作几乎都是内在局部的，这对于减少工作集的规模有着巨大的意义。

2.3 现代计算机架构的特征

为了让大型程序高效运作，程序员心中必须大致明白程序是怎样运作的。这并不意味着每个程序员要成为架构师和编译器专家——一个能够表达系统重要的性能方面的工作模型就已经足够了。

2.3.1 面向性能的架构

性能和效率一直是计算机硬件制造商最关心的内容，同时在现代计算架构中存在的大量技巧就是为了直接解决性能问题。

内存子系统。近几十年，计算机内存经常被组织成多级架构，这种多级架构是一种逐渐减少容量以增加带宽和减少延时的妥协方案。除了主存之外，几乎每个现代计算机至少有两级缓存，现在三级缓存也开始变得更加普遍。内存子系统在多核和多嵌套字系统中的性能也变得更加重要。

对于越来越深的内存体系结构来说，主存带宽和计算能力间越来越大的差距是造成深度内存架构的部分原因，同时，工作集和缓存容量在性能中所扮演的角色也是无法忽视的。

虚拟内存存在现代系统中也是相当普遍的，现代系统也支持不同大小的页。硬件对页的支持已经让大多数代码无须考虑 TLB 的容量，但是一定要注意有些特定情况。

线程级并行。在过去的十年中，几乎每个主流处理器开始增多其内核的数量，同时，多嵌套字配置使得工作站和服务器间的共享内存并行比以前更加容易实现。

通过直接增加控制无关路径和功能单元的数量，多核和多嵌套字为提高性能提供了相似的机制。硬件支持的同时多线程有时会引发混乱——那些被单核所支持的“硬件线程”使得代码在单核上比较容易使用所有资源，但是多线程的出现并没有增加单核上设施的可用性。

数据级并行。虽然和 20 世纪 80 年代流行的向量处理器有着相似之处，但是 SIMD 指令集通常基于全宽（full-width）功能单元并且受益于现在硬件中的很多超标量特征。SIMD 通常在“包装”好的输入数据上执行，但不利于控制分歧。

指令级并行。现代硬件具有高度超标量体系结构，这种体系结构使用很多不一样的处理器，这些处理器的特征有乱序执行，多执行端口，精细的预取，预测机制，以及分支预测硬件。这种并行通常不会被程序员所看到，但是没有这些特征将会带来损失（由每周期执行的指令测量）并且可能预示着资源使用率的缺失。

2.3.2 编程工具和运行时

对于程序员幸运的是，我们所使用的工具——编译器、调试器和采样分析器，随着硬件变得越来越精细，越来越强大。OpenMP 已经发展到了 4.0，这使得它成为一个开发多线程程序的选择。

本章不使用 OpenMP 的细节技术，只是简单让 OpenMP 在运行时管理线程的数量和其同类之间关系的细节。对于规模可变的实验中，设置环境变量 `PMP_NUM_THREADS` 用来管理线程数量是非常有用的，同时设置 `KMP_AFFINITY` 的散播或者集中可以用来控制 SMT 应该如何使用。注意，`KMP_AFFINITY` 是 Intel 编译器的量，对于 gcc 使用 `GOMP_CPU_AFFINITY`。

对于向量化，有多种可行的方法。最简单的是出现在许多现代编译器中的自动向量化技术。它的目的是基于少量代码产生高效的为 SIMD 代码。用自动向量分析器转换和生产高效代码的方式编写代码是一种艺术，也是最令人满意的方法。

编译器内置函数（intrinsics）是表达向量化代码最可靠的方式，这些内置的数据类型和函数直接翻译成 SIMD 寄存器和指令。然而，它们是由供应商定制的，有些会晦涩难懂。

一个替代内置函数的选择是 C++ 类（这种方法在本章中使用最频繁），这个类包装内置函数。这些类很容易自己编写——Intel 编译器带有一组——它们提供一个方便的抽象层，这样可以为 SIMD 宽度提供一些灵活性。甚至可以重载标准算术运算符，这使得使用它们编码的方式和使用标准 C++ 标量类型编码类似。

2.3.3 计算环境

Intel xeon E5-2680 处理器。它是基于 x86 架构的新一代众核服务器架构多核处理器。这种处理器有超标量，乱序执行的内核，这种内核支持两路超线程。除了标量单元之外，它还有一个 256 位宽的 SIMD 单元（用于执行 AVX 指令集）。分离的乘法和加法端口允许一条加法指令和一条乘法指令（每个都 4 位宽，双精度）在一个周期中完成。本章考虑的是 2 嵌套字、16 核配置。

Intel Xeon Phi 协处理器。这种处理器的一个重要特征是单个裸芯片上有很多按序执行内核。每个内核支持 4 通道多线程，这种结构能够隐藏多周期指令和内存访问的延迟。协处理器中的内核是顺序执行的，每个周期都发出不超过一个向量指令，同时运行的频率比处理器低。这些内核有特殊的 51 位宽的向量指令集，同时支持融合乘加（FMA），这使它们能够执行 8 位宽双精度乘加操作。本章使用的是预生产的芯片。

协处理器在物理上安装在 PCIe 卡上，同时使用了 GDDR 内存和 Linux 操作系统。本章中，我们要原生地在协处理器上运行实验——二进制和输入要与卡共享，并且完全运行在这上面。

更多的信息在图 2-3 上表示。一级和二级缓存的大小对于处理器和协处理器来说都是针对每个内核的，三级缓存对于处理器是针对每个内核的，但是在片上所有内核之间是共享的。

	Intel Xeon Processor E5-2680	Intel Xeon Phi 协处理器
套接字×内核×SMT时钟(GHz)	2 × 8 × 2	1 × 60 × 4
每个内核每周期执行的指令数	2.7	1.09
单精度GFLOP/s	2	2
双精度GFLOP/s	691	2127
	346	1063
L1 / L2 / L3 缓存(KB)	32 / 256 / 20,480	32 / 512 / -
DRAM	128 GB	4 GB GDDR
STREAM 中的带宽	76 GB/s	150 GB/s

图 2-3 系统配置

2.4 通向高性能的路

本节描述如何将以性能为核心的特性模型应用到计算环境中以加速 Hydro2D。首先简单浏览参考代码。

2.4.1 运行 Hydro2D

Hydro2D 是使用 Godunov 方法求解欧拉方程的一种实现。给定一个离散的初始化边界值问题，它沿着从 $t_{\text{initial}}=0$ 到 t_{final} 的计算步长推进。

这段代码首先由一系列的参数文件所构成，读取这些文件用来形容要解决的问题，同时告诉这个程序在执行的时候应该怎样表现（关于输出、阻塞等）。

线程的数量根据 OpenMP 运行时确定。下面的示例使用两线程运行，默认相关配置的情况（见图 2-4）。

命令的输出是为了告知用户参数的选择和求解器处理的进度。

.nml 文件并没有完全包含整个要解决的离散问题。取而代之的是，Hydro2D 代码有很多用户可以指定顺序生成的测试问题，用户只需要指定测试问题的尺寸和数量（见图 2-5）。


```
[user@localhost $] OMP_NUM_THREADS=2 ./hydro -i input_problem.nml
+-----+
|nx=250|
|ny=125|
|nxystep=125|
|tend=200.000|
|nstepmax=1000000|
|noutput=0|
|dtoutput=2.000|
+-----+
Lower corner test case : 2 2
Hydro starts.
Hydro: OpenMP mode ON
Hydro: OpenMP 2 max threads
Hydro: OpenMP 1 num threads
Hydro: OpenMP 4 num procs
Hydro: standard build
HydroC: Simple decomposition
HydroC: nx=1 ny=1
--> step= 1, 1.33631e-03, 1.33631e-03 {1034.06 Mflops 20307925 Ops} (0.020s)
--> step= 2, 2.67261e-03, 1.33631e-03 {1048.12 Mflops 19401667 Ops} (0.019s)
--> step= 3, 5.17799e-03, 2.50538e-03 {1241.47 Mflops 20307925 Ops} (0.016s)
--> step= 4, 7.68338e-03, 2.50538e-03 {1150.96 Mflops 19401667 Ops} (0.017s)
...
```

图 2-4 运行参考 Hydro2D 代码的例子

```
This namelist contains various input parameters for HYDRO runs

&RUN
tend=50
#noutput=10
nstepmax=1133
dtoutput=2.
/

&MESH
nx=256
ny=256
nxystep=125
prt=0
dx=0.05
boundary_left=1
boundary_right=1
boundary_down=1
boundary_up=1
testcase=1
/

&HYDRO
courant_factor=0.8
niter_riemann=10
/
```

图 2-5 Hydro2D 文件的输入样例

2.4.2 Hydro2D 的结构

Hydro2D 参考代码有近 5000 行，分布在 19 个 header/c 文件对中，但是核心计算例程非常紧凑。

计算方案

在每个时间步长中会执行下列步骤，以此推进到下一个时间步长所需要的解：

更新。根据维度分裂方法，更新可以在 $x \rightarrow y/y \rightarrow x$ 维度模式中选择；每次更新包含单元格的流量计算（实际上就是在每个垂直于更新方向上的单元格中构建和求解黎曼问题）。在更新中，所有黎曼问题都是独立的。当一个单元格的流量计算出来时，时间上的积分就可以运算了。

时间步长计算。为了积分的稳定性，必须基于每个单元格中的状态计算和归约 courant 数。这是一个归约过程。

数据结构

Hydro2D 的数据结构非常接单。为了简化一些边界计算，解状态存储在一个由两个单

元格组成的普通单元格“halo层”中。网格被组织成 $[\rho, \rho u, \rho v, E]^T$ 的形式, 变量都存储于各自的平面中。这是使用的是 y-major 空间维度排序。因为考虑到问题的对称性, x-major 和 y-major 是等价的。

除了全局问题状态之外, 参考版本的 Hydro2D 代码有一个次级数据结构用于存放中间状态的值。

slab。Hydro2D 参考代码工作在单独的多层网格平台上, 相比于全局网格, 这些网络会有更小的维度。这种结构叫作 slab, 它允许格格不入的计算并作为构造和屏蔽结构。

在每个更新步中, 一部分单元格复制进 slab 中。然后每个更新都在 slab 中进行, 这些 slab 分别存储子域中每个交界处/单元格的中间态。当积分完成时, 结果被写回结果网格, 同时另外一个子区域的值复制进 slab 中, 直到整个解网格更新结束。从图 2-6 可以看到整个过程。

值得注意的是, 使用参考代码中 x 和 y 维度上的更新都能用 slab, 为了妥善处理边界值, 更新 x 行和 y 列在并没有得到副本前完成。对于 y 轴来说, 复制进/出 slab 的数据是原始数据的转置, 因此 slab 总是必须足够宽以容纳网格的 x 和 y 维度 (另外一个 slab 维度是用户可选择的参数)。

关键函数。对于这份参考代码中有 11 个关键函数; 有些函数的实现很简单。这些函数都可以在自己的源文件中找到。简单的总结如下。

- `compute_deltat()`: 这个函数用来计算在局部黎曼问题中的最大特征值, 此特征值将在接下来的计算中求解最大稳定时间步长 (`compute_deltat.c:159`)。
- `hydro_godunov()`: 这是更新操作中的最高层例程, 其调用几乎所有的下级函数以复制数据到 slab, 计算更新, 写回。这个函数的一个参数控制哪一个维度将会被遍历 (`hydro_godunov.c:62`)。
- `gatherConservativeVars()`: 将解从子区域解网格复制到 slab 的 conservation 变量 (`conserver.c:48`) 中存储。
- `constoprim()`: (部分地) 将 slab 中 conservation 变量转变为流量计算所需要的基本形式 (`constoprim.c:48`)。
- `equation_of_state()`: 使用 `constoprim()` 函数的结果加上 conservative 变量中保存的 E 项以完成最后初始的变量 p , 在 slab 的专用存储中完成这里所有操作 (`equation_of_state.c:48`)。
- `slope()`: 检查临接单元格的值, (以基本的形式) 计算 (或限制) 斜率。此斜率将用于接下来的流量计算。这份代码使用了 Leer 的 MC 限制器 (`slope.c:52`)。
- `trace()`: 将计算出来的斜率应用到每个单元格上的终止点上 (`trace.c:51`)。
- `qlleftright()`: 将 `trace()` 的结果复制于独立缓冲区上, 为 `riemann()` 做准备 (`qlleftright.c:48`)。
- `riemann()`: `riemann()` 函数是流量计算的核心; 这个函数是一个用于解决中间态压力的标量非线性系统。这个函数用 Newton-Raphson 方法进行求解。余下的术语是从压力计算中产生的 (`riemann.c:80`)。
- `cmpflx()`: 使用 `Riemann()` 计算的中间态和周围状态计算差值, 并转变为 conservation 流量 (`cmpflx.c:51`)。
- `updateConservativeVars()`: 使用邻接的流量来积分单元格, 同时将结果复制

回全局解网格 (conservar.c:122) 中。

slab 例程 `gatherConservativeVars()` 通过 `updateconservative-vars()` 能够使用“块同步并行”方式相继操作。这些函数能够组织成无数据相关性的形式（对于每个函数使用输入/输出存储），所以每个函数都暴露出线程和数据并行的可能性（通过 OpenMP 和自动向量化）。这就是计算这些块的地方，同时 `hydro_godunov()` 是这些单元格的调用框架。图 2-7 展示了在一维切片上计算时这些结构和数据流的情况。

`compute_deltat()` 类似地使用 slab 去计算一系列中间状态（包括 `equation_of_state()`），这些中间状态最终要得到最大特征值，这些特征值是归约得到的。这些 slab 步具有与上述相同的线程级和数据级并行性。

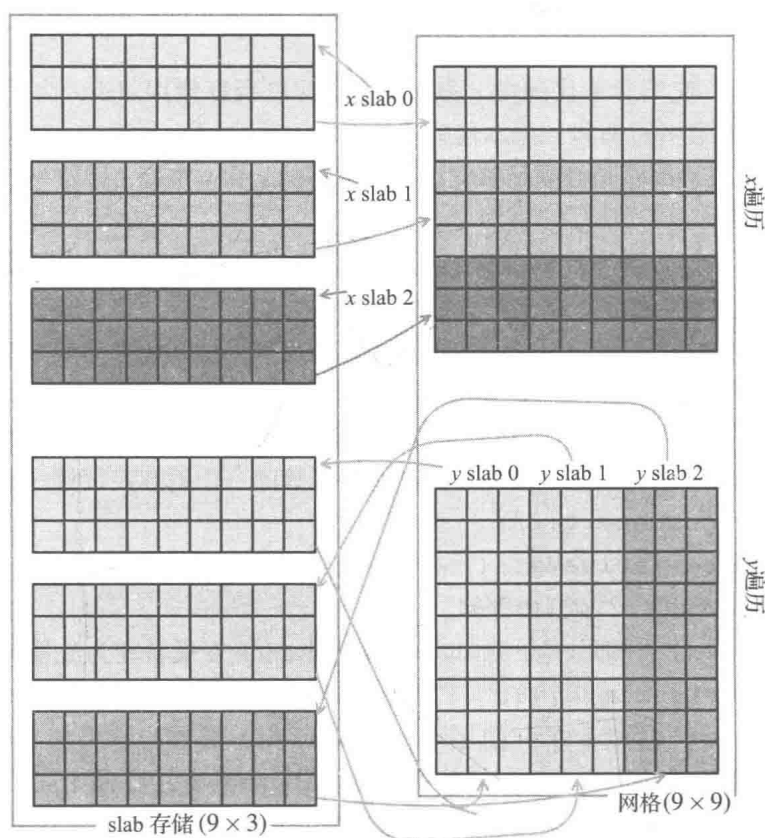


图 2-6 这张图展示了基于 slab 的 Godunov 调度方法是如何在 slab 上移动并计算数据的。注意，图中的列的“复制”只是在表达时间序列而不是真的复制——这里一个 slab 和一个网格参与运算

测试性能

当然，本章的目的是提升 Hydro2D 代码的性能而不是进行新的科学实验。我们最关注的是某些特定性能的值，这些性能需要能够代表在实验中程序的运行情况。

对于我们的程序来说，测量这些值非常简单：Hydro2D 的性能很大程度上独立于初始值和边界条件的值，所以我们不必要对测试问题做出特定的限制。尽管在黎曼问题求解器中使用的 Newton-Raphson 迭代有控制流的出现，因为这些流量的计算会依赖于输入，所以这些控制流会增加流量计算的运行时间，但是这个只对极端情况有意义。

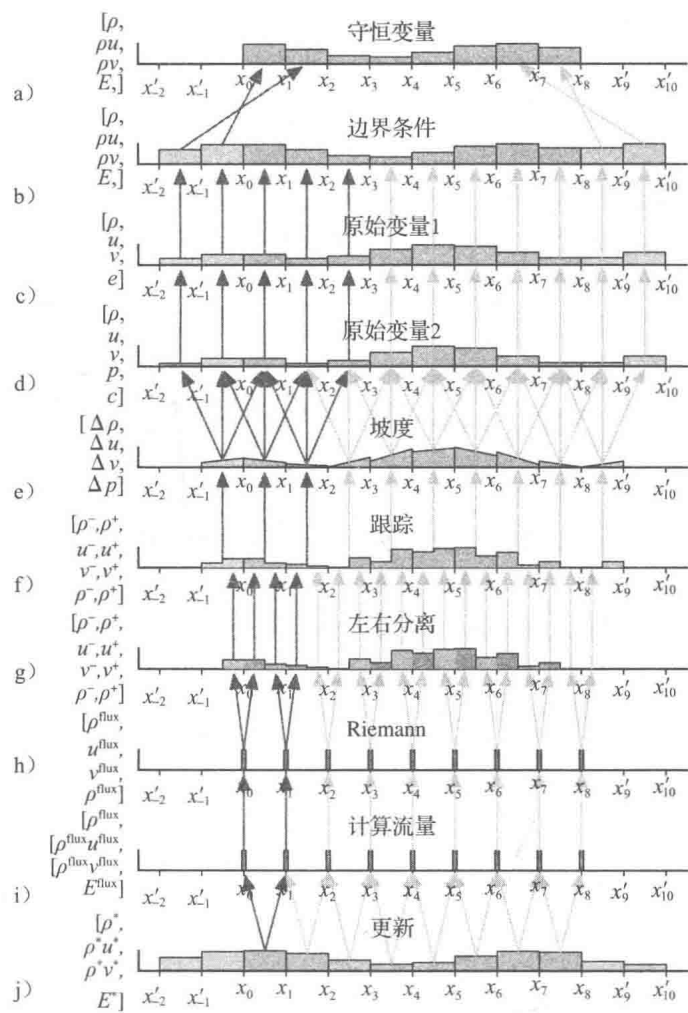


图 2-7 1D 坡度限制 Godunov 方法的步骤和访问模式。浅色箭头标明数据在每步间网格的依赖性，深色箭头表示第一个单元格 (x_1, x_0) 间的依赖性

为了了解代码对问题规模的敏感度，我们将探索不同规模的问题，并且正规化我们的实验结果到每个单元格在每个时间步长所花的时间。每步所花的时间由稳定性条件所表述，而所谓稳定性条件就是解状态和网格空间的乘积。这就意味着两个不同的初始条件可能需要不同的步数以达到给定时间的仿真。

图 2-8 展示了参考代码中单元格步 / 秒的性能。处理器的性能是协处理器的两倍多，协处理器的并行效率非常差。

2.4.3 优化

尽管支持并行，但原始代码的性能非常糟糕。就像我们将要展示的，程序效率低的部分原因是一些代码中高级的、概念

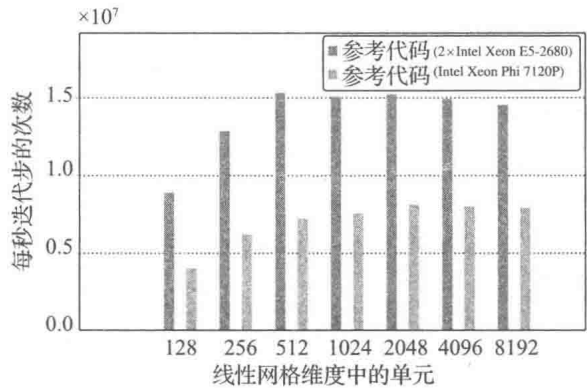


图 2-8 Hydro2D 参考代码在处理器和协处理器上每个网格步 / 秒上的性能

性的结构在运行的时候效率不高，也因为代码转化的效率并不好。就像加一句 `#pragma omp parallel for` 并不保证循环的并行执行效率，`#pragma vector` 也并不一定意味着高效的向量化。

我们将展示这份代码中的不足之处同时也将想办法改进这些不足。

优化代码通常所需要注意的事项。这里展示的优化代码将原来代码的结构和格式与原始代码做了一些分离。这些改变并没直接影响性能，这将会让代码的清晰性和可维护性得到提高。

- 文件结构重新整理过：核心例程位于 `pcl-hydro-core.cpp` 和 `pcl-hydro-vcore.cpp` 中，同时实用例程移到了 `pcl-hydro-util.cpp` 和 `pcl-hydro-params.cpp` 中。最高层的驱动写入 `run-tilde.cpp` 中。
- 为了提高清晰性，一些标识符已重命名（即，`constoprim()` 重命名为 `conservative_to_primitive()`）。`equation_of_state()` 分为 `equation_of_state()` 和 `speed_of_sound()`，以此来区分它们的不同用途。
- `REAL_T` 被引入到代码中，此类型代替原有 `single` 和 `double` 类型。
- 在参考代码 `Hydro2D` 中，大多数函数中使用 `idim` 参数以控制调整数据访问（特别是，在当前维度方向上， ρu 、 ρv 必须分别对待）。在优化代码中，多数分支被如下方式所取代：按正确顺序把未知项的数组按 $(\rho, \rho u, \rho v, E)$ 传递给被调用函数，而不是使用模糊的多维数组 `u`。
- 占位符函数 `my_sqrt()` 和 `rcp()` 被 `sqrt` 和 `1/<x>` 所代替，这样在选择执行何种操作时，提供了更加有弹性的选择。
- 原始代码中的一些常数也被重写，比如 `Hgamma` 被 `GAMMA` 所替代。
- 栈分配变量和函数参数尽可能以 `const` 修饰。

2.4.4 内存使用

在原始的代码中，`slab` 方法使用的类 GPU 内存模型，这种方法并没充分利用现代系统的内存层次结构。

从 `slob` 复制进和复制出数据的开销并不能与其他计算重叠，而且这种操作计算密度较低。更糟糕的是，`slab` 计算使用了很多中间变量，这些中间变量急剧提升了内存占用率。这些问题有时可以通过中间存储的重用得到缓解，但是在每次更新中对小区域的依赖性表明应该使用不同的方法解决这个问题。

完善 `compute_deltat()`。独立计算每个时间步长会导致性能下降，除此之外，因为安全地进入和退出遍历需要使用同步，所以它需要重读整个网格并降低速度；这也会成为巨大的损失。

事实上，时间步长的计算只在计算开始阶段和在奇数步中遍历 x 轴后才会发生。除第一次之外，`compute_deltat()` 函数将读取的数据和奇数步后 x 轴遍历最后一部分积分的数据是相同的，积分后计算决定每个单元格中最大传播速度，而最大传播速度决定了固定时间步长。

全局最大速度的归约的计算可以作为 `work-sharing` 的一部分，`work-sharing` 是在每次更新步中都要做的工作。在遍历时每个线程独立地计算并本地归约所需要的最大速度，同时时间步循环可以在线程专有的值上协调一个全局归约。如果通信成本占了主要消耗，此操作可使用树形结构实现。

通过结合 `compute_deltat()` 和积分步, 可减少网格遍历和同步的花销。

旋转更新。图 2-7 中垂直箭头展示了流量计算和积分在步间的依赖性; 对于计算网格的一维的行 (或者列), 这里既有重用, 但也有依赖性的“窗口”。

沿着每个网格 strip 通过使用“旋转”以完全重用中间计算和最小化中间存储是可行的。

- 首先, 更新中的全部的 slab 变量被“收缩”, 即, `constoprime()/conservative_to_primitive()`、`slope()`、`trace()`、`riemann()`、`cmpflx()` 和 `updateConservativevars()/update()`; `qllefttright()` 除了复制数据并没有真正工作, 我们将其移除。这些函数同时在一个单元格中运行; 同时这些函数使用子问题所需要的最少输入集合, 并且通过输出参数返回结果的向量值。
- 在 strip 更新开始的时候, priming 阶段使用边界条件 (通过调用 `set_boundaries()`), 即装载合适的 conservative 变量, 接着计算决定左边界流量所需要的数据 (即, 图 2-7 中在 x_0 处的流量) ——所有有用的中间值保存在环形缓冲区中, 这些值可能要用于之后网格的更新, 同时所有其他中间值被舍弃。参考 `strip_prime()` (图 2-10, 参考图 2-9 中的描述) 实现这部分代码。
- 之后, 为了更新 strip 中所有网格, 在 stable 阶段, 加载环形缓冲区存储的数值和 strip 中接下来需要的 conservative 变量, 接着计算下一次流量计算所需要中间值。前一步中的流量和最新计算的流量在图 2-7j 阶段结合在一起, 用于更新当前的网格, 同时把未来 stable 步所需要的中间值放置进了环形缓冲区中。参考 `strip_stable()` (见图 2-11)。

图 2-7 表明, 在图 2-7i 中对 x_i 上单流量的依赖最宽的点是图 2-7d 阶段中第二轮 primitive 变量的运算——最后结合四个值算出结果。因此没有环形缓冲区包含多于四个值, 这就保持了中间存储的成本较低。

```
1 struct strip_work
2 {
3     REAL_T flux      [4][2]; // flux at i-1/2, i+1/2
4     REAL_T left_flux [4][2]; // left_flux at i, i+1
5     REAL_T prim      [5][3]; // prim for i, i+1, i+2
6 };
```

图 2-9 在 `strip_prime()` 和 `strip_stable()` 中旋转更新结构

2.4.5 线程级并行

在参考代码中, 线程级并行只发生在 slab 的 y 维度上 (或者“高”维度) (回想图 2-6), 这对并行有很大限制。

slab 调度策略中在存储、通信和负载均衡上的消耗表明原始方案并不如意, 需要改进。我们将参考代码转换为称为块 (tiling) 的域分解方法。参考图 2-12, 这是一个图片解释。

仿真网格沿着 x 和 y 维度被切割成一系列矩形小块, 这些小块被分配给线程。这个调度方案并不要求一对一的线程与块的映射, 但是我们现有的实现只支持一对一的情况。每个小块对于它的子域、维度、stride 和其他需要记录的信息, 有一套独立分配的一套解和变量 (参见图 2-13)。

halo 区。每个小块维护了四个 halo 区 (在 x 和 y 方向上分别有两个 halo 块), 这些区

域的宽度是2，这有利于提升块与块间的通信效率和应用边界条件（将这个条件应用到和原始网格共享边的块上）。这就是在块中的 $\{x, y\}_{y, var} \text{stride}$ 和 $\{x, y\}_{edges}$ 域。

```

1 void strip_prime(strip_work *restrict sw,
2                 const REAL_T *restrict rho,
3                 const REAL_T *restrict rho_u,
4                 const REAL_T *restrict rho_v,
5                 const REAL_T *restrict E,
6                 const hydro *restrict h,
7                 const int stride,
8                 const REAL_T dtdx)
9 {
10     REAL_T pre_prim[5][4]; // i-2, i-1, i, i+1
11     for(int i = 0; i < 4; ++i)
12     {
13         const int src_offs = (- 2 + i)*stride;
14         REAL_T E_internal;
15         conservative_to_primitive(pre_prim[0] + i, pre_prim[4] + i, pre_prim[1] + i,
16                                 pre_prim[2] + i, &E_internal,
17                                 rho[src_offs], rho_u[src_offs], rho_v[src_offs], E[src_offs]);
18         pre_prim[3][i] = equation_of_state(pre_prim[0][i], E_internal);
19     }
20     REAL_T pre_dvar[4][2]; // i-1, i
21     if(h->iorder != 1)
22         for(int i = 0; i < 2; ++i)
23             for(int v = 0; v < 4; ++v)
24                 pre_dvar[v][i] = slope(pre_prim[v][0+i], pre_prim[v][1+i],
25                                       pre_prim[v][2+i], h->slope_type, h->inv_slope_type);
26     REAL_T pre_left_flux [4][2]; // i-1, i
27     REAL_T pre_right_flux[4][2]; // i-1, i
28     for(int i = 0; i < 2; ++i)
29     {
30         const REAL_T prim_c = speed_of_sound(pre_prim[4][i + 1], pre_prim[3][i+1]);
31         trace(pre_left_flux [0] + i, pre_left_flux [1] + i,
32              pre_left_flux [2] + i, pre_left_flux [3] + i,
33              pre_right_flux[0] + i, pre_right_flux [1] + i,
34              pre_right_flux[2] + i, pre_right_flux[3] + i,
35              pre_prim[0] [i+1], pre_prim[4][i+1], pre_prim[1] [i+1],
36              pre_prim[2] [i+1], pre_prim[3] [i+1],
37              pre_dvar[0] [i], pre_dvar[1] [i],
38              pre_dvar[2] [i], pre_dvar[3] [i],
39              prim_c, rcp(prim_c),
40              dtdx);
41     }
42     REAL_T gdnv_rho, gdnv_u, gdnv_v, gdnv_p;
43     riemann(&gdnv_rho, &gdnv_u, &gdnv_v, &gdnv_p,
44            pre_left_flux [0][0], pre_left_flux [1][0], pre_left_flux [2][0],
45            pre_left_flux [3][0],
46            pre_right_flux[0][1], pre_right_flux [1][1], pre_right_flux [2][1],
47            pre_right_flux [3][1]);
48     cmpflx(sw->flux[0] + 0, sw->flux[1] + 0, sw->flux[2] + 0, sw->flux[3] + 0,
49            gdnv_rho, gdnv_u, gdnv_v, gdnv_p);
50     for(int v = 0; v < 4; ++v)
51         sw->left_flux[v][0] = pre_left_flux[v][1];
52     for(int v = 0; v < 5; ++v)
53     {
54         sw->prim[v][0] = pre_prim[v][2];
55         sw->prim[v][1] = pre_prim[v][3];
56     }
57 }

```

图 2-10 旋转更新起始函数。参考图 2-9 对 strip_work() 函数的表述


```

1 REAL_T strip_stable(const hydro *restrict h,
2                     REAL_T      *restrict rho,
3                     REAL_T      *restrict rhou,
4                     REAL_T      *restrict rhov,
5                     REAL_T      *restrict E,
6                     strip_work *restrict sw,
7                     const int   i,
8                     const int   stride,
9                     const REAL_T dtdx,
10                    const bool   do_courant)
11 {
12     const int src_offs = (i + 2)*stride;
13
14     REAL_T E_internal;
15     conservative_to_primitive(sw->prim[0] + 2, sw->prim[4] + 2, sw->prim[1] + 2,
16                               sw->prim[2] + 2, &E_internal,
17                               rho[src_offs], rhou[src_offs],
18                               rhov[src_offs], E[src_offs]);
19     sw->prim[3][2] = equation_of_state(sw->prim[0][2], E_internal);
20
21     REAL_T dvar[4]; // slope for i+1
22     if(h->iorder != 1)
23         for(int v = 0; v < 4; ++v)
24             dvar[v] = slope(sw->prim[v][0], sw->prim[v][1], sw->prim[v][2], h->
25                             slope_type, h->inv_slope_type);
26
27     REAL_T right_flux[4];
28     const REAL_T prim_c = speed_of_sound(sw->prim[4][1], sw->prim[3][1]);
29     trace(sw->left_flux [0] + 1, sw->left_flux [1] + 1, sw->
30           left_flux [2] + 1, sw->left_flux [3] + 1,
31           right_flux + 0, right_flux + 1, right_flux
32           + 2, right_flux + 3,
33           sw->prim[0] [1], sw->prim[4][1], sw->prim[1] [1], sw->prim
34           [2] [1], sw->prim[3] [1],
35           dvar[0], dvar[1], dvar[2],
36           dvar[3],
37           prim_c, rcp(prim_c),
38           dtdx);
39
40     REAL_T gdnv_rho, gdnv_u, gdnv_v, gdnv_p;
41     riemann(&gdnv_rho, &gdnv_u, &gdnv_v, &
42            gdnv_p,
43            sw->left_flux [0][0], sw->left_flux [1][0], sw->left_flux [2][0], sw->
44            left_flux [3][0],
45            right_flux[0], right_flux[1], right_flux[2],
46            right_flux[3]);
47
48     cmpflx(sw->flux[0] + 1, sw->flux[1] + 1, sw->flux[2] + 1, sw->flux[3] + 1,
49            gdnv_rho, gdnv_u, gdnv_v, gdnv_p);
50
51     const REAL_T new_rho = update(rho [i*stride], sw->flux[0][0], sw->flux[0][1],
52                                   dtdx);
53     const REAL_T new_rhou = update(rhou[i*stride], sw->flux[1][0], sw->flux[1][1],
54                                    dtdx);
55     const REAL_T new_rhov = update(rhov[i*stride], sw->flux[2][0], sw->flux[2][1],
56                                    dtdx);
57     const REAL_T new_E = update(E [i*stride], sw->flux[3][0], sw->flux[3][1],
58                                  dtdx);
59
60     REAL_T courantv = (REAL_T) 0.0;
61     if(do_courant)
62     {
63         REAL_T prim_rho, prim_inv_rho, prim_u, prim_v, E_internal;
64         conservative_to_primitive(&prim_rho, &prim_inv_rho, &prim_u, &prim_v, &
65                                   E_internal,
66                                   new_rho, new_rhou, new_rhov,
67                                   new_E);
68         const REAL_T prim_p = equation_of_state(prim_rho, E_internal);
69         const REAL_T prim_c = speed_of_sound (prim_inv_rho, prim_p);

```

图 2-11 旋转更新稳定状态函数。参考 2-9 对 strip_work() 函数的表述


```

54     courant(&courantv, prim_u, prim_v, prim_c);
55 }
56
57 rho [i*stride] = new_rho;
58 rhov[i*stride] = new_rhov;
59 rhov[i*stride] = new_rhov;
60 E   [i*stride] = new_E;
61
62 for(int v = 0; v < 4; ++ v)
63 {
64     sw->flux      [v][0] = sw->flux      [v][1];
65     sw->left_flux[v][0] = sw->left_flux[v][1];
66 }
67 for(int v = 0; v < 5; ++ v)
68 {
69     sw->prim[v][0] = sw->prim[v][1];
70     sw->prim[v][1] = sw->prim[v][2];
71 }
72
73 return courantv;
74 }

```

图 2-11 (续)

每个块和它们自己的网格(有更小的维度)有相同的运行模式;更新时使用与其所在网格相同的顺序;同时每个块分别独立计算 x 轴和 y 轴遍历。当一个遍历运算完成时,有必要通过使用 `send_{low,high}_{x,y}_edge()` 函数使用最新的积分数据更新它们临近的单元格。在 x 轴遍历开始之前, x halo 必须被交换。同样,在 y 轴遍历开始前, y halo 也必须被交换。时间步长计算作为 x 轴遍历在奇数时间步中的一部分将在本地完成,而且规约要在所有的块间实现(见图 2-14 ~ 图 2-16)。

减少通信。为了最小化通信开销和块间的伪分享问题,我们分配每个 halo 区时,让它们与自己和其他块不共享缓存块。然后,在每个通信阶段,每个块将复制它自己的边域(宽度是 2)给对应的邻居。

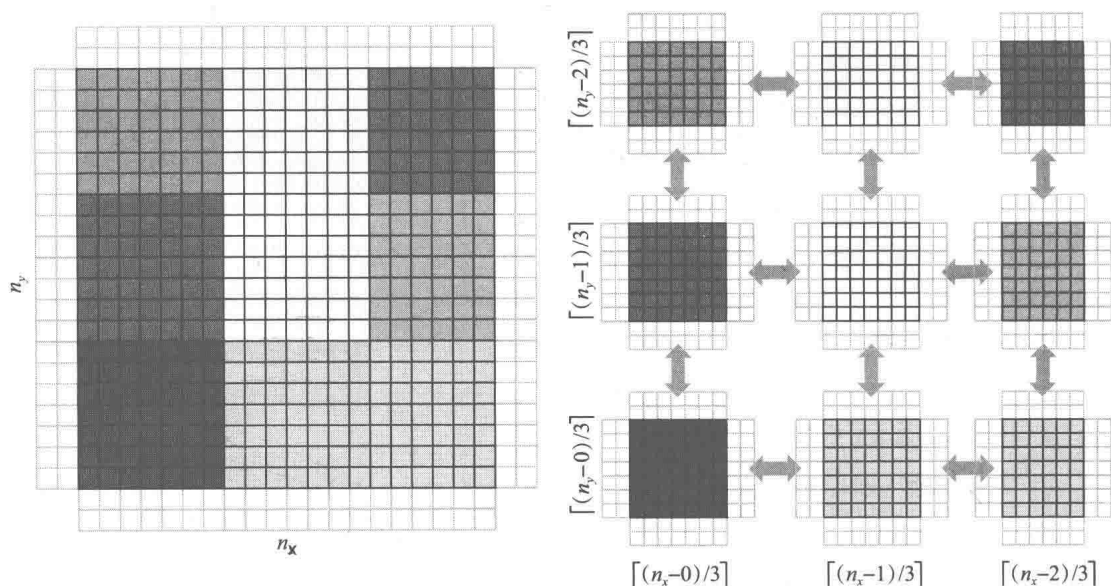


图 2-12 块方法: 整个区域被分割成网格状的子区域, 每个子区域都和原来一样运作(除了 halo 区(或者 ghost 区), 这些区域和邻近块的更新信息形成分解的内部填充)


```

1 struct hydro_decomp
2 {
3     struct tile
4     {
5         int    position[2];
6         tile *neighbors[4]; // w, e, s, n
7
8         int offset[2];
9         int n[2];
10
11        int ystride;
12        int varstride;
13
14        int    x_e_ystride;
15        int    x_e_varstride;
16        REAL_T *x_edges[2];
17        int    y_e_ystride;
18        int    y_e_varstride;
19        REAL_T *y_edges[2];
20
21        REAL_T *q;
22        REAL_T *rho;
23        REAL_T *rho;
24        REAL_T *rho;
25        REAL_T *E;
26    };
27
28    int    decomp[2];
29    int    ntiles;
30    tile *tiles;
31 };

```

图 2-13 Hydor 计算块的数据结构

```

1 REAL_T x_step(hydro_decomp *hd, const hydro *restrict h, const REAL_T dtdx, int
   tid, bool do_courant)
2 {
3     return tile_x_step(hd->tiles + tid, dtdx, h, 0, hd->tiles[tid].n[1],
   do_courant);
4 }
5
6 REAL_T tile_x_step(tile *restrict tl, const REAL_T dtdx, const hydro *restrict h,
   int jstart, int jend, bool do_courant)
7 {
8     static const REAL_T x_signs[4] = {1.0, -1.0, 1.0, 1.0};
9
10    REAL_T courantv = 0.0;
11    for(int j = jstart; j < jend; ++j)
12    {
13        const int ob = (j + 2) * tl->ystride + 0;
14        if(tl->neighbors[0])
15        {
16            for(int v = 0; v < 4; ++v)
17            {
18                tl->q[v*tl->varstride + ob + 0] = tl->x_edges[0][v*tl->
   x_e_varstride + j * tl->x_e_ystride + 0];
19                tl->q[v*tl->varstride + ob + 1] = tl->x_edges[0][v*tl->
   x_e_varstride + j * tl->x_e_ystride + 1];
20            }
21        }
22        else
23            set_boundaries(tl->q + ob, x_signs, 2, 1, 4, tl->

```

图 2-14 x_strp 和 tile_x_step; 执行给定块 x 轴遍历的代码。同时包含了
归约 / 返回时间步长计算遇到的最大特征值


```

        varstride);
24
25     if(tl->neighbors[1])
26     {
27         for(int v = 0; v < 4; ++v)
28         {
29             tl->q[v*tl->varstride + ob + tl->n[0] + 2 + 0] = tl->x_edges[1][v*
                tl->x_e_varstride + j * tl->x_e_ystride + 0];
30             tl->q[v*tl->varstride + ob + tl->n[0] + 2 + 1] = tl->x_edges[1][v*
                tl->x_e_varstride + j * tl->x_e_ystride + 1];
31         }
32     }
33     else
34         set_boundaries(tl->q + ob + tl->n[0] + 3, x_signs, 2, -1, 4, tl->
            varstride);
35
36     const int o = ob + 2;
37     strip_work sw;
38     strip_prime(&sw, tl->rho + o, tl->rhou + o, tl->rhov + o, tl->E + o, h,
        1, dtdx);
39
40     for(int i = 0; i + 1 <= tl->n[0]; ++i)
41     {
42         const REAL_T cv = strip_stable(h, tl->rho + o, tl->rhou + o, tl->rhov
            + o, tl->E + o, &sw, i, 1, (REAL_T) dtdx, do_courant);
43         courantv = std::max(courantv, cv);
44     }
45 }
46
47 return courantv;
48 }

```

图 2-14 (续)

```

1 void y_step(hydro_decomp *hd, const hydro *restrict h, const REAL_T dtdx, int
    tid)
2 {
3     tile_y_step(hd->tiles + tid, dtdx, h, 0, hd->tiles[tid].n[0]);
4 }
5
6 void tile_y_step(tile *restrict tl, const REAL_T dtdx, const hydro *restrict h,
    int istart, int iend)
7 {
8     static const REAL_T y_signs[4] = {1.0, 1.0, -1.0, 1.0};
9
10    for(int i = istart; i + 1 <= iend; ++i)
11    {
12        const int ob = i + 2;
13        if(tl->neighbors[2])
14        {
15            for(int v = 0; v < 4; ++v)
16            {
17                tl->q[v*tl->varstride + ob ] = tl->y_edges[0][v*tl
                    ->y_e_varstride + i];
18                tl->q[v*tl->varstride + 1*tl->ystride + ob] = tl->y_edges[0][v*tl
                    ->y_e_varstride + 1*tl->y_e_ystride + i];
19            }
20        }
21        else
22            set_boundaries(tl->q + ob, y_signs, 2, tl
                ->ystride, 4, tl->varstride);
23
24        if(tl->neighbors[3])
25        {
26            for(int v = 0; v < 4; ++v)
27            {
28                tl->q[v*tl->varstride + (2 + tl->n[1])*tl->ystride + ob] = tl->

```

图 2-15 y_step() 和 tile_y_step(); 代码用于执行一个块的 y 轴遍历


```

29         y_edges[1][v*tl->y_e_varstride + i];
        tl->q[v*tl->varstride + (3 + tl->n[1])*tl->ystride + ob] = tl->
            y_edges[1][v*tl->y_e_varstride + 1*tl->y_e_ystride + i];
30     }
31 }
32 else
33     set_boundaries(tl->q + ob + (tl->n[1] + 3)*tl->ystride, y_signs, 2, -
        tl->ystride, 4, tl->varstride);
34
35     const int o = ob + 2 * tl->ystride;
36     strip_work sw;
37     strip_prime(&sw, tl->rho + o, tl->rhov + o, tl->rhou + o, tl->E + o, h,
        tl->ystride, dtdx);
38     for(int j = 0; j < tl->n[1]; ++j)
39         strip_stable(h, tl->rho + o, tl->rhov + o, tl->rhou + o, tl->E + o, &
            sw, j, tl->ystride, dtdx, false);
40 }
41 }

```

图 2-15 (续)

线程必须明确它们之间的通信以协调 halo 区域的交换和计算固定时间步长；为了尽可能避免每次遍历的全局同步，每个线程在一个块上工作时，这个块只通过点对点的同步与其他邻居块上工作的线程同步。甚至固定时间步长的计算中全局归约可以用避免通信的树形归约来实现。

```

1  hydro H;
2  load_hydro_params(&H, input_file, quiet);
3  const int nthreads = omp_get_max_threads();
4
5  init_hydro(&H);
6  hydro_decomp HD;
7  init_hydro_decomp(&HD, &H, nthreads, quiet);
8
9  REAL_T dt = compute_timestep(&H) / (REAL_T) 2.0;
10 REAL_T dt_dx = dt/H.dx;
11 set_scheme(H.scheme, dt_dx);
12 // align and pad shared courantv array to avoid false sharing
13 REAL_T *courantv = (REAL_T*) _mm_malloc(64 * nthreads, 64);
14 const int cache_stride = 64/sizeof(REAL_T);
15
16 #pragma omp parallel
17 {
18     const int tid = omp_get_thread_num();
19     barrier.init(tid);
20     send_low_x_edge (HD.tiles + tid);
21     send_high_x_edge(HD.tiles + tid);
22     send_low_y_edge (HD.tiles + tid);
23     send_high_y_edge(HD.tiles + tid);
24     barrier.wait(tid);
25     while(H.step < H.nstepmax && H.t < H.tend)
26     {
27         barrier.wait(tid);
28         if(tid == 0)
29             H.t += dt;
30
31         if(H.step % 2 == 0)
32         {
33             send_low_x_edge (HD.tiles + tid);
34             send_high_x_edge(HD.tiles + tid);
35             barrier.wait(tid);
36             x_step(&HD, &H, dt_dx, tid, false);
37             barrier.wait(tid);

```

图 2-16 块化 Hydro2D 代码的时间步长循环


```

38         send_low_y_edge (HD.tiles + tid);
39         send_high_y_edge(HD.tiles + tid);
40         barrier.wait(tid);
41         y_step(&HD, &H, dt_dx, tid);
42     }
43     else
44     {
45         send_low_y_edge (HD.tiles + tid);
46         send_high_y_edge(HD.tiles + tid);
47         barrier.wait(tid);
48         y_step(&HD, &H, dt_dx, tid);
49         barrier.wait(tid);
50         send_low_x_edge (HD.tiles + tid);
51         send_high_x_edge(HD.tiles + tid);
52         barrier.wait(tid);
53         courantv[tid*cache_stride] = x_step(&HD, &H, dt_dx, tid, true);
54         barrier.wait(tid);
55         if(tid == 0)
56         {
57             // serial reduction among threads; tree scheme suitable for
58             // large number of threads
59             for(int i = 1; i < nthreads; ++i)
60                 courantv[0] = std::max(courantv[i*cache_stride], courantv[0]);
61             dt = H.courant_number * H.dx / courantv[0];
62             dt_dx = dt/H.dx;
63             set_scheme(H.scheme, dt_dx)
64         }
65     }
66     barrier.wait(tid);
67 }

```

图 2-16 (续)

点对点同步和树形归约对于交叉开关阵列和少数的多核系统的影响不是那么大，但是全局栅栏的代价是随着处理器的数量以 $O(\log n)$ 的量级增长的，随着越来越多的内核搭载到处理器上，其性能可能将会因为非一致内存访问（NUMA）而受到更大的影响。交换缓存和 halo 区是有存储代价的，不过对于一个大规模网格来说这些代价是微不足道的，甚至，在一个 240 线程的协处理器上这些消耗也是非常小，同时，相比于 slab 所需要的大规模中间存储，这些消耗也很小。

继续优化。刚刚提到的优化结果（相比于参考代码）在图 2-17 中呈现。相对于图 2-8 中参考代码的性能，这展示了对于处理器和协处理器使用固定时间步，旋转更新和块的应用之后，优化后代码的性能（用时间作为参考量）。到此为止，处理器的性能提高了 2 ~ 3.3 倍和协处理器的性能提高了 2.3 ~ 3.6 倍。这表明在这个计算环境中，合理使用线程和内存子系统可以在代码和架构上有更多收益。就像我们马上要看到的，我们将会得到更多有益的东西。

2.4.6 算术效率和指令级并行

现代处理器的微架构是非常精细的，但掌握一定基础后就可以依照此进行优化。

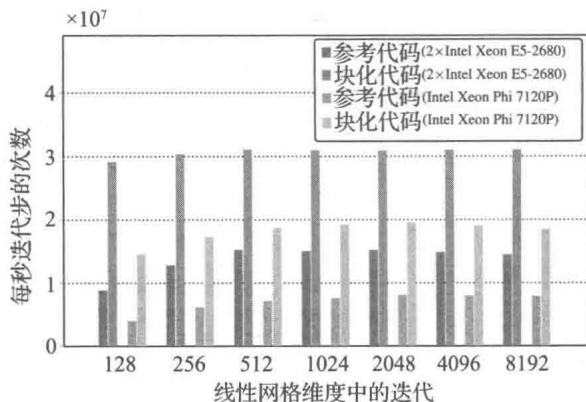


图 2-17 参考代码的性能和线程 / 访存改进后的性能

记住。除法和超越数学函数对计算机而言代价是很高的,即便当直接被指令集支持时,也是这样,它们通常会比乘法和除法慢 1~2 个数量级。这份参考 Hydro2D 代码使用了很多除法和开方操作,这些操作占据了程序大量的运行时间。硬件中执行这些操作的功能单元可成为运算流水线的瓶颈,它会限制吞吐量和指令级并行。

因为这些操作如此昂贵,同时一些平方根和除法要重复多次计算,所以一些经常使用的值可以之前就计算好。在原始代码中, constoprime(), equation_of_state() 和 trace() 都是 ρ 和 c (声音的本地速度) 相除许多次的操作。可以计算它们的倒数并把结果记录下来用于之后的流量计算。除此之外,相对于除法,使用倒数和乘法在现代操作系统中更快。要知道很多编译器出于精度考虑不会自动做这样的转换。

```

1 int goon = 1;
2 int iter;
3 for (iter = 0; iter < Hniter_riemann ; iter++) {
4     if (goon) {
5         double wwl, wwr;
6         wwl = sqrt(cl_i * (one + gamma6 * (pstar_i - pl_i) / pl_i));
7         wwr = sqrt(cr_i * (one + gamma6 * (pstar_i - pr_i) / pr_i));
8         double ql = two * wwl * Square(wwl) / (Square(wwl) + cl_i);
9         double qr = two * wwr * Square(wwr) / (Square(wwr) + cr_i);
10        double usl = ul_i - (pstar_i - pl_i) / wwl;
11        double usr = ur_i + (pstar_i - pr_i) / wwr;
12        double delp_i = MAX((qr * ql / (qr + ql) * (usl - usr)), (-pstar_i));
13        pstar_i = pstar_i + delp_i;
14        // Convergence indicator
15        double uo_i = DABS(delp_i / (pstar_i + smallpp));
16        goon = uo_i > PRECISION;
17    }
18 }

```

图 2-18 参考 riemann() 中 Newton-Raphson 循环的核心

分解。Hydro2D 代码最核心的部分 (Newton-Raphson 迭代, 位于 riemann.c 中) 有 8 次除法; 参考图 2-18。使用迭代中简单的代数操作, 我们能够减少这些操作的数量。检查迭代的更新 Δp^* , 我们将会看到表达式 (从图 2-18 中第 15 行开始)。

$$\Delta p^* = \frac{q_l q_r (u_l^* - u_r^*)}{(q_r + q_l)} \quad (2-3a)$$

其中,

$$q_l = \frac{2w_l'^3}{w_l'^2 + c_l} \quad (2-3b)$$

$$c_l = \gamma p_l \rho_l \quad (2-3c)$$

$$u_l^* = u_l - \frac{p^* - p_l}{w_l'} \quad (2-3d)$$

$$q_r = \frac{2w_r'^3}{w_r'^2 + c_r} \quad (2-3e)$$

$$c_r = \gamma p_r \rho_r \quad (2-3f)$$

$$u_r^* = u_r + \frac{p^* - p_r}{w_r'} \quad (2-3g)$$

(这里, r 是媒介的隔热率, 在该方法中是一个常数。) 结合式 (2-3a) 和式 (2-3b)~(2-3g), 我们能够减少总的除法数量:

$$\begin{aligned}
\Delta p^* &= \frac{\frac{2w_r'^3}{w_r'^2 + c_r} \frac{2w_l'^3}{w_l'^2 + c_l} \left(u_l - u_r - \frac{p^* - p_r}{w_r'} - \frac{p^* - p_l}{w_l'} \right)}{\frac{2w_r'^3}{w_r'^2 + c_r} + \frac{2w_l'^3}{w_l'^2 + c_l}} \\
&= \frac{\frac{4w_r'^3 w_l'^3}{(w_r'^2 + c_r)(w_l'^2 + c_l)} \left(u_l - u_r - \frac{p^* - p_r}{w_r'} - \frac{p^* - p_l}{w_l'} \right)}{\frac{2w_r'^3 (w_l'^2 + c_l) + 2w_l'^3 (w_r'^2 + c_r)}{(w_l'^2 + c_l)(w_r'^2 + c_r)}} \quad (2-4) \\
&= \frac{2w_r'^3 w_l'^3 \left(u_l - u_r - \frac{p^* - p_r}{w_r'} - \frac{p^* - p_l}{w_l'} \right)}{w_r'^3 (w_l'^2 + c_l) + w_l'^3 (w_r'^2 + c_r)} \\
&= \frac{2w_r'^2 w_l'^2 Z}{w_r'^3 (w_l'^2 + c_l) + w_l'^3 (w_r'^2 + c_r)}
\end{aligned}$$

其中,

$$Z = w_r' w_l' (u_l - u_r) - w_l' (p^* - p_r) - w_r' (p^* - p_l)$$

这个等式消除了中间值 $q_{l,r}$ 和 $u_{l,r}^*$ 同时没有引入额外的除法。最终我们能够结合 w_{lr}' 与式 (2-3c) 从而避免除法。

$$\begin{aligned}
w_l' &= \sqrt{c_l \left(1 + \frac{\gamma+1}{2\gamma} \frac{p^* - p_l}{p_l} \right)} \\
&= \sqrt{c_l + \frac{\gamma+1}{2} \rho_l (p^* - p_l)} \quad (2-5)
\end{aligned}$$

对 w_r' 使用相似的表达式; 使用我们重写的式 (2-4) 和式 (2-5), 我们又减少了 Newton-Raphson 迭代中除法 / 求倒数的数量, 从原先的 8 个减少到两个。参考图 2-19 中优化的代码。

进一步优化性能。使用结合了旋转更新的线程级 / 内存分块方法, 以及结合了数值计算方法提升的 Courant/ 更新方法后, 对于程序性能的提升会在图 2-20 中展示。我们看到我们的使用了特定的指令和计算使得处理器性能提升了 1.2 ~ 1.7 倍, 协处理器提升了 1.4 ~ 1.6 倍。再一次, 使用提升计算环境的原则和完全一样的代码在两种架构上得到了相似的加速比。

2.4.7 数据级并行

数据级并行可以巨大地提升性能, 这种收益与借助向量化减少的指令数量相关——简单地说, 我们将向量化代码, 尽可能让这份代码尽可能看上去像串行代码。

原始的代码依赖于编译器指令实现向量化, 但是它糟糕的性能意味着程序中有大量的指令开销, 编译器并不总是能够推测出程序员在代码中所表达的想法。

我们使用 C++ SIMD 类来实现流量计算和积分运算中的向量化, 这是一种性能和开发工作量的合理妥协。使用这种方法可以清晰表达出如何向量化, 而且不需要考虑内置函数的细节和汇编代码所带来的阻碍。


```

1 for(int i = 0; i < NITER_RIEMANN; ++i)
2 {
3     if(goon)
4     {
5         const REAL_T left_ww2 = left_rho * ((REAL_T) 0.5) * ((GAMMA + (REAL_T)
6             1.0) * p_star + (GAMMA - (REAL_T) 1.0) * left_p);
7         const REAL_T left_ww = my_sqrt(left_ww2);
8         const REAL_T right_ww2 = right_rho * ((REAL_T) 0.5) * ((GAMMA + (REAL_T)
9             1.0) * p_star + (GAMMA - (REAL_T) 1.0) * right_p);
10        const REAL_T right_ww = my_sqrt(right_ww2);
11        const REAL_T tmp_num = ((REAL_T)2.0) * left_ww2 * right_ww2 * (left_ww *
12            right_ww * (left_u - right_u) - left_ww * (p_star - right_p) -
13            right_ww * (p_star - left_p));
14        const REAL_T tmp_den = right_ww2*right_ww * (left_ww2 + left_c) +
15            left_ww2*left_ww * (right_ww2 + right_c);
16        const REAL_T tmp = tmp_num * rcp(tmp_den);
17        const REAL_T deleft_p = std::max(tmp, -p_star);
18
19        p_star += deleft_p;
20
21        const REAL_T uo = std::abs(deleft_p * rcp(p_star + SMALLPP));
22        goon = uo > PRECISION;
23    }
24 }

```

图 2-19 优化代码中(分解的) Newton-Raphson 循环的核心

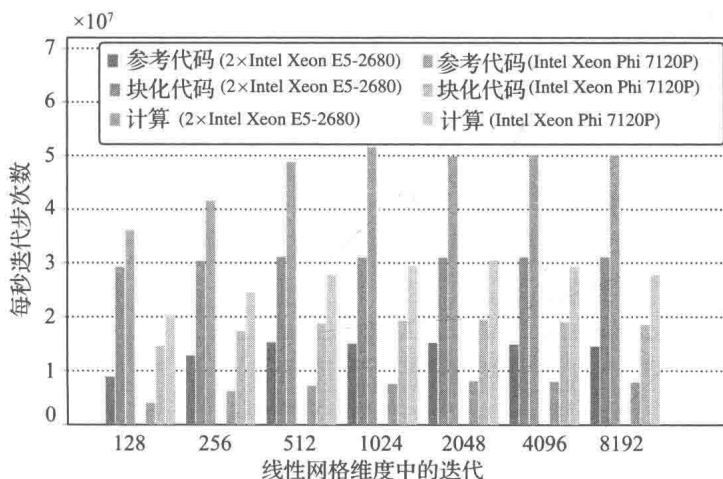


图 2-20 参考代码的性能, 线程 / 访存优化后的性能, 数值计算优化后的性能

新代码使用宏 `SIMD_WIDTH` 作为底层硬件的 SIMD 宽度的占位符 (对于处理器是 2 或者 4, 对于协处理器是 8)。

SIMD 2 路。参考代码使用的 slab 方法保证了所有的计算使用相同的数据布局 (总是在 slab 的 x 方向)。由于该方法中 x、y 遍历的不同, 本地更新和“旋转更新”求解的方法在这两个遍历上的行为是不同的。图 2-21 描述了这种方法中不同的 SIMD 方案。

为了实现上述想法, Intel 编译器在 `<dvec.h>` 头文件中提供了 C++ SIMD 类。每个内核使用适当的 SIMD 类型 (我们使用 `VREAL_T` 宏来简化) 来进行并行化, 同时对于例程的核本身来说这些在形式上几乎没有改变。

`vstrip_prime()` 和 `vstrip_stable()` 函数与自身的串行代码几乎相同。而 `strip_prime()` 和 `strip_stable()` 在进行 x 和 y 遍历时行为是相同的 (除了步 (stride) 以及 `pu` 和 `pv` 变量的顺序), 但 x 的更新要求沿着数据布局的方向进行遍历, 这要求我们引入一个

`hstrip_stable()` 函数以稍微不同的方式处理循环缓冲区。这在图 2-21c 展示；另一种方案是像原始方案一样像 y 轴坐标一样重排 x 轴的数据，如图 2-21b 所示。

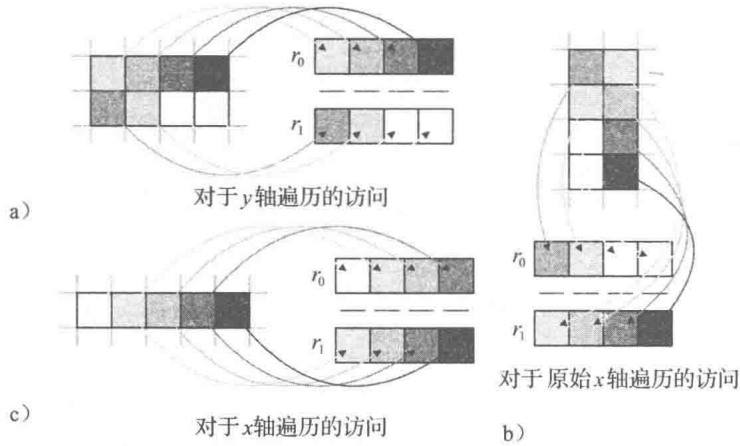


图 2-21 向量化方案：将网格中的数据移动到 SIMD 寄存器的策略（用 r_0 和 r_1 标明）。对于 y 轴遍历 a) 来说，向量化是十分容易的，因为数据在内存中和寄存器上有着相同的存储方法。对于 x 轴遍历 b) 来说，向量需要额外的一次转置。另外一种可选的方案 c) 可以避免转置而且消耗的资源也不多。途中暗色的条是指计算器正在计算这些网格中的流量

图 2-22 展示新的 x 遍历更新函数；其主要新特征是使用 `rotate_left_wml()`、`rotate_left_wm2()` 函数，这个对函数分别用 1、2 改变第一个参数通道里的内容并使用最左边通道的第二个参数替换新“空”通道的内容。图 2-23 显示了 `tile_x_step()` 函数如何应用于 SIMD。我们省略了新 `tile_y_step()` 的描述，可以预见的是，这将类似于新的 `tile_x_step()`，除了边界的初始化以上使用 `vstrip_prime()` 和 `vstrip_stable()` 以外，它们将完全相同。

```

1 VREAL_T hstrip_stable(const hydro *restrict h,
2                       REAL_T      *restrict rho,
3                       REAL_T      *restrict rhou,
4                       REAL_T      *restrict rhov,
5                       REAL_T      *restrict E,
6                       vstrip_work *restrict sw,
7                       const int    i,
8                       const int    stride,
9                       const VREAL_T dtdx,
10                      const VMASK_T write_mask,
11                      const bool    do_courant)
12 {
13     const int src_offs = (i + 2)*stride;
14
15     VREAL_T E_internal;
16     vconservative_to_primitive(sw->prim[0] + 2, sw->prim[4] + 2, sw->prim[1]
17                               + 2, sw->prim[2] + 2, &E_internal,
18                               loadu(rhou + src_offs), loadu(rhov + src_offs), loadu(E +
19                               src_offs));
20     sw->prim[3][2] = vequation_of_state(sw->prim[0][2], E_internal);
21     for(int v = 0; v < 5; ++v)
22     {
23         rotate_left_wm2(sw->prim[v] + 0, sw->prim[v][2]);
24     }
25 }

```

图 2-22 `hstrip_stable():strip_stable()` 的一个向量变体，这是为了向量化 x 遍历更新而不聚集 / 扩散


```

23     rotate_left_wmi(sw->prim[v] + 1, sw->prim[v][2]);
24 }
25
26 VREAL_T dvar[4];    // slope for i+1
27 if(h->iorder != 1)
28     for(int v = 0; v < 4; ++ v)
29         dvar[v] = vslope(sw->prim[v][0], sw->prim[v][1], sw->prim[v][2], (
30             VREAL_T) h->slope_type, (VREAL_T) h->inv_slope_type);
31
32 VREAL_T right_flux[4];
33 const VREAL_T prim_c = vspeed_of_sound(sw->prim[4][1], sw->prim[3][1]);
34 vtrace(sw->left_flux [0] + 1, sw->left_flux [1] + 1,
35     sw->left_flux [2] + 1, sw->left_flux [3] + 1,
36     right_flux + 0, right_flux + 1,
37     right_flux + 2, right_flux + 3,
38     sw->prim[0] [1], sw->prim[4][1], sw->prim[1] [1], sw->prim
39     [2] [1], sw->prim[3] [1],
40     dvar[0], dvar[1], dvar[2],
41     dvar[3],
42     prim_c, rcp(prim_c),
43     dtdx);
44
45 for(int v = 0; v < 4; ++v)
46     rotate_left_wmi(sw->left_flux[v] + 0, sw->left_flux[v][1]);
47
48 VREAL_T gdnv_rho, gdnv_u, gdnv_v, gdnv_p;
49 vriemann(&gdnv_rho, &gdnv_u, &gdnv_v,
50     &gdnv_p,
51     sw->left_flux [0][0], sw->left_flux [1][0], sw->left_flux [2][0], sw
52     ->left_flux [3][0],
53     right_flux[0], right_flux[1], right_flux[2],
54     right_flux[3]);
55
56 vcmpflx(sw->flux[0] + 1, sw->flux[1] + 1, sw->flux[2] + 1, sw->flux[3] + 1,
57     gdnv_rho, gdnv_u, gdnv_v, gdnv_p);
58
59 for(int v = 0; v < 4; ++v)
60     rotate_left_wmi(sw->flux[v] + 0, sw->flux[v][1]);
61
62 const VREAL_T new_rho = vupdate(load(rho + i*stride), sw->flux[0][0],
63     sw->flux[0][1], VREAL_T(dtdx));
64 const VREAL_T new_rhou = vupdate(load(rhou + i*stride), sw->flux[1][0],
65     sw->flux[1][1], VREAL_T(dtdx));
66 const VREAL_T new_rhov = vupdate(load(rhov + i*stride), sw->flux[2][0],
67     sw->flux[2][1], VREAL_T(dtdx));
68 const VREAL_T new_E = vupdate(load(E + i*stride), sw->flux[3][0],
69     sw->flux[3][1], VREAL_T(dtdx));
70
71 VREAL_T courantv = (VREAL_T) 0.0;
72 if(do_courant)
73 {
74     VREAL_T prim_rho, prim_inv_rho, prim_u, prim_v, E_internal;
75     vconservative_to_primitive(&prim_rho, &prim_inv_rho, &prim_u, &prim_v,
76         &E_internal,
77         new_rho, new_rhou, new_rhov,
78         new_E);
79     const VREAL_T prim_p = vequation_of_state(prim_rho, E_internal);
80     const VREAL_T prim_c = vspeed_of_sound (prim_inv_rho, prim_p);
81     vcourant(&courantv, prim_u, prim_v, prim_c, write_mask);
82 }
83
84 maskstore(rho + i*stride, new_rho, write_mask);
85 maskstore(rhou + i*stride, new_rhou, write_mask);
86 maskstore(rhov + i*stride, new_rhov, write_mask);
87 maskstore(E + i*stride, new_E, write_mask);
88
89 for(int v = 0; v < 4; ++ v)
90 {
91     sw->flux [v][0] = sw->flux [v][1];
92     sw->left_flux[v][0] = sw->left_flux[v][1];
93 }

```

图 2-22 (续)


```

79     }
80     for(int v = 0; v < 5; ++ v)
81     {
82         sw->prim[v][0] = sw->prim[v][2];
83         sw->prim[v][1] = sw->prim[v][2];
84     }
85
86     return courantv;
87 }

```

图 2-22 (续)

剥落和掩码。当考虑在任意网格维度上 SIMD 时，我们一定要看如何处理工作中的“剩余”部分（即，循环末尾的工作并没有填充完 SIMD 单元的数据）。不适当处理会导致硬件和软件在运行时出错。通常的方法是循环剥离——主循环随着 SIMD 的步伐而推进，一个单独、串行的循环处理剩下的东西。这种方法会带来一些额外的开销，特别是主循环循环长度减少、SIMD 宽度增长的时候，但对于大网格，这样操作 SIMD 所带来的收益会变少。

另一种方法需要使用支持写掩码的指令集，比如 Intel Xeon Phi 协处理器。这里，一个位掩码决定了向量处理器的哪个通道被写进内存，哪些被跳过。使用这种结构，就有可能避免循环剥离。这种机制在处理 SIMD 处理器不同寄存器要表现不同行为的时候使用。

```

1  REAL_T tile_x_step(tile *restrict tl, const REAL_T dtdx, const hydro *restrict h,
2      int jstart, int jend, bool do_courant)
3  {
4      static const REAL_T x_signs[4] = {1.0, -1.0, 1.0, 1.0};
5      VMASK_T alltrue;
6      mask_true(&alltrue);
7      VINT_T linear;
8      linear_offset(&linear);
9
10     VREAL_T courantv      = (VREAL_T) 0.0;
11     REAL_T  final_courantv = 0.0;
12     for(int j = jstart; j < jend; ++j)
13     {
14         const int  ob = (j + 2) * tl->ystride + 0;
15         if(tl->neighbors[0])
16         {
17             for(int v = 0; v < 4; ++v)
18             {
19                 tl->q[v*tl->varstride + ob + 0] = tl->x_edges[0][v*tl->
20                     x_e_varstride + j * tl->x_e_ystride + 0];
21                 tl->q[v*tl->varstride + ob + 1] = tl->x_edges[0][v*tl->
22                     x_e_varstride + j * tl->x_e_ystride + 1];
23             }
24         }
25         else
26             set_boundaries(tl->q + ob, x_signs, 2, 1, 4, tl-> varstride);
27
28         if(tl->neighbors[1])
29         {
30             for(int v = 0; v < 4; ++v)
31             {
32                 tl->q[v*tl->varstride + ob + tl->n[0] + 2 + 0] = tl->x_edges[1][v*
33                     tl->x_e_varstride + j * tl->x_e_ystride + 0];
34                 tl->q[v*tl->varstride + ob + tl->n[0] + 2 + 1] = tl->x_edges[1][v*
35                     tl->x_e_varstride + j * tl->x_e_ystride + 1];
36             }
37         }
38     }
39     else

```

图 2-23 向量版的 tile_x_step；与图 2-14 不同的是，SIMD_WIDTH 更新步长，使用了掩码产生 hstrip_stable，并且循环剥离在 SIMD 交界处减小了 Courant 值


```

35     set_boundaries(tl->q + ob + tl->n[0] + 3, x_signs, 2, -1, 4, tl->
        varstride);
36
37     const int o = ob + 2;
38     strip_work sw;
39     strip_prime(&sw, tl->rho + o, tl->rhov + o, tl->rhov + o, tl->E + o, h,
        1, dtdx);
40
41     vstrip_work vsw;
42     for(int v = 0; v < 4; ++ v)
43     {
44         vsw.flux [v][0][SIMD_WIDTH-1] = sw.flux [v][0];
45         vsw.left_flux [v][0][SIMD_WIDTH-1] = sw.left_flux [v][0];
46     }
47     for(int v = 0; v < 5; ++ v)
48     {
49         vsw.prim [v][0][SIMD_WIDTH-2] = sw.prim [v][0];
50         vsw.prim [v][0][SIMD_WIDTH-1] = sw.prim [v][1];
51         vsw.prim [v][1][SIMD_WIDTH-1] = sw.prim [v][1];
52     }
53
54     int i = 0;
55     for(; i + SIMD_WIDTH <= tl->n[0]; i+=SIMD_WIDTH)
56     {
57         const VREAL_T cv = hstrip_stable(h, tl->rho + o, tl->rhov + o, tl->
            rhov + o, tl->E + o, &vsw, i, 1, (VREAL_T) dtdx, alltrue,
            do_courant);
58         courantv = std::max(courantv, cv);
59     }
60
61     for(; i < tl->n[0]; i+=SIMD_WIDTH)
62     {
63         const VMASK_T in_bounds = mask_lt(linear + (VINT_T) (double)i, (VINT_T)
            (double)tl->n[0]);
64         const VREAL_T cv = hstrip_stable(h, tl->rho + o, tl->rhov + o,
            tl->rhov + o, tl->E + o, &vsw, i, 1, (VREAL_T) dtdx, in_bounds,
            do_courant);
65         courantv = std::max(courantv, cv);
66     }
67 }
68
69
70 for(int i = 0; i < SIMD_WIDTH; ++i)
71     final_courantv = std::max(final_courantv, courantv[i]);
72 return final_courantv;
73 }

```

图 2-23 (续)

协处理器对每个通道掩码有最基本的支持。几乎每个向量指令都会接收一个掩码参数，同时，存储和操作这些掩码都会有专门的掩码寄存器。处理器为了达到这种效果必须混合指令，这意味着需要结合和交叉两个寄存器的内容。这就刺激了指令数量的增长。

控制分支。对于有效向量化来说，更加深远的挑战是控制分支——SIMD 分支代码的处理。这个问题会在有数据依赖程序执行时产生，同时在 SIMD 中解通常会执行所有的分支，使用混合 / 掩码操作以保证结果的正确性。在运行较多代码时，SIMD 的指令的收益会随之减少。

Godunov 方法利用 Riemann 求解器抛出异常的方法避免控制分支；Newton-Raphson 迭代不必一致收敛。一些输入可能要求更多的迭代，而这个会造成 SIMD 的控制分支问题。参见图 2-24 中向量化后的循环。

基于我们的观察，Riemann 求解器中的 Newton-Raphson 迭代几乎没有出现控制分支的问题；参考代码对于求解器在迭代上有用户定义的限制，但是 99% 的 Riemann 计算收敛前

只需要一次迭代。

对齐。数据对齐的代价会随着架构而不同，并且它们的影响严重依赖于 SIMD 操作的加载部分。在这份代码增加的部分中，对齐“块”分配非常简单；对于一些网格尺寸，填充行（可以使得网格的 0 列保持对齐）是非常有必要的。图 2-25 展示了优化实现中的初始化代码。

最后一步。通过减少中间储存“块”和算术优化的方法显示的算术强度的提升已经被我们的向量化策略实现。图 2-26 显示了每次增量优化后的性能，加上参考代码，每一部分都使得处理器加速了 1.4 ~ 1.5 倍，协处理器加速了 2.2 ~ 4.4 倍！在这一点，多亏了宽向量单元，使得协处理器得以改进，但是对于小尺寸问题，协处理器的表现并不优秀。协处理器有较高吞吐量时效率才会突出。

```

1   for(int i = 0; i < NITER_RIEMANN && !all_zero(goon); ++i)
2   {
3       const VREAL_T left_ww2 = left_rho * ((VREAL_T) 0.5) * (VREAL_T(GAMMA + (
4           REAL_T) 1.0) * p_star + VREAL_T(GAMMA - (REAL_T) 1.0) * left_p);
5       const VREAL_T left_ww = my_sqrt(left_ww2);
6       const VREAL_T right_ww2 = right_rho * ((VREAL_T) 0.5) * (VREAL_T(GAMMA + (
7           REAL_T) 1.0) * p_star + VREAL_T(GAMMA - (REAL_T) 1.0) * right_p);
8       const VREAL_T right_ww = my_sqrt(right_ww2);
9       const VREAL_T tmp_num = ((VREAL_T)2.0) * left_ww2 * right_ww2 * (left_ww
10          * right_ww * (left_u - right_u) - left_ww * (p_star - right_p) -
11          right_ww * (p_star - left_p));
12       const VREAL_T tmp_den = right_ww2*right_ww * (left_ww2 + left_c) +
13          left_ww2*left_ww * (right_ww2 + right_c);
14       const VREAL_T tmp = tmp_num * rcp(tmp_den);
15       const VREAL_T deleft_p = std::max(tmp, -p_star);
16
17       p_star += select_true(goon, deleft_p, (VREAL_T) 0.0);
18
19       const VREAL_T uo = std::abs(deleft_p * rcp(p_star + SMALLPP));
20       goon = mask_and(goon, mask_gt(uo, VREAL_T(PRECISION)));
21   }

```

图 2-24 vriemann() 中向量化的 Newton-Raphson 循环；注意 all_zero() 调用，这造成了所有 SIMD 通道在产生分歧前执行

2.5 总结

我们为了求解 2D 欧拉方程而实现的改进的 Godunov 方法极大地提升了参考代码的性能，无论对于协处理器还是处理器，无论问题的规模大小，它们的性能都得到了提升。

2.5.1 协处理器与处理器

在参考代码中，协处理器只有原先代码一半的效率——从原始 FLOP 能力看（参考图 2-3）——这完全出乎意料。优化的代码扭转了这个情况，协处理器的性能是处理器的 1.3 ~ 1.5 倍。在 Intel Xeon Phi 协处理器上的并行化反映出它需要较大的问题规模才能达到性能峰值，但给足工作量，它能轻松超过处理器的性能。

2.5.2 水涨船高

我们做的每个优化都让程序得到很大收益，无论从处理器还是从协处理器的角度看都是这样，对于问题的规模，性能也是提升的——在协处理器上有 12 倍的性能提升，处理器上有 5 倍的性能提升。虽然两个不同计算机架构在很多方面都不同，但是它们提升性能的方式

根本上是相同的, 而且这些优化同样也会在其他机器上起作用。

```

1 inline unsigned long long round_to_alignment(unsigned long long x, int alignment)
2 {
3     if(x & (alignment-1))
4         x = (x & ~(alignment-1)) + alignment;
5     return x;
6 }
7
8 void init_tile(tile *tl, int xstart, int xend, int ystart, int yend)
9 {
10     static const int target_alignment = 64;
11     static const int arith_alignment = target_alignment/sizeof(REAL_T);
12
13     tl->offset[0] = xstart;
14     tl->offset[1] = ystart;
15
16     tl->n[0] = xend - xstart;
17     tl->n[1] = yend - ystart;
18
19     const int min_stride = tl->n[0] + 2*2;
20     tl->ystride = round_to_alignment(min_stride, arith_alignment);
21     tl->varstride = tl->ystride * (tl->n[1] + 2*2);
22
23     const int alloc_offset = arith_alignment - 2;
24     tl->q = ((REAL_T *) _mm_malloc( sizeof(REAL_T) * (4 * tl->varstride +
        arith_alignment), target_alignment)) + alloc_offset;
25     tl->rho = tl->q + 0*tl->varstride;
26     tl->rhov = tl->q + 1*tl->varstride;
27     tl->rhov = tl->q + 2*tl->varstride;
28     tl->E = tl->q + 3*tl->varstride;
29
30     const int min_x_e_varstride = tl->n[1] * 2;
31     tl->x_e_ystride = 2;
32     tl->x_e_varstride = round_to_alignment(min_x_e_varstride, arith_alignment);
33     tl->x_edges[0] = (REAL_T *) _mm_malloc( sizeof(REAL_T) * 4 * tl->x_e_varstride
        , target_alignment);
34     tl->x_edges[1] = (REAL_T *) _mm_malloc( sizeof(REAL_T) * 4 * tl->x_e_varstride
        , target_alignment);
35
36     const int min_y_e_varstride = tl->n[0] * 2;
37     tl->y_e_ystride = tl->n[0];
38     tl->y_e_varstride = round_to_alignment(min_y_e_varstride, arith_alignment);
39     tl->y_edges[0] = (REAL_T *) _mm_malloc( sizeof(REAL_T) * 4 * tl->y_e_varstride
        , target_alignment);
40     tl->y_edges[1] = (REAL_T *) _mm_malloc( sizeof(REAL_T) * 4 * tl->y_e_varstride
        , target_alignment);
41 }

```

图 2-25 初始化函数 `inti_tile()` 和辅助函数 `round_to_alignment()`、`_mm_malloc()` 是一个编译器内置函数, 用于对齐分配。由于 halo 区的关系, 理想的 SIMD 访问内存不能够在分配开始的地方进行网格的内存分配, 这份代码每行的起始非 halo 区将排成行, 这要求 `alloc_offset` 提前分配指针

2.5.3 性能策略

科学家还有所有关注代码性能的编码者, 都非常关注如何最小化优化程序所需要的改变和维护代价。在这个主题上, 没有简单的答案——没有“银弹”。

当本章深入分析的时候, 并不是旨在声称每个程序员是微体系结构的专家, 或者他们熟悉目标处理器的微小结构。相反, 我们希望通过有意识地忽略和简化硬件, 显著改善效率, 这样减少了有些处理器独有硬件特征对优化代码所带来的影响。

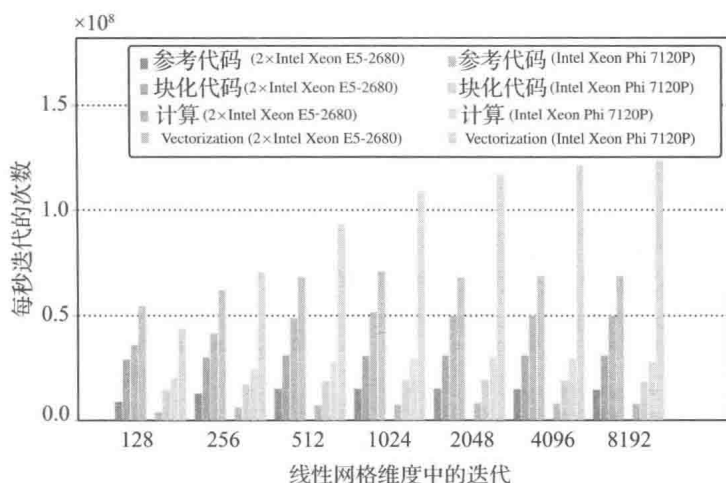


图 2-26 参考代码的性能，线程 / 访存优化后的性能，数值计算优化后的性能，向量化后的性能

- 按照我们所描述算术优化，考虑如何在给定架构上执行等价的不同数学变换。这些变换经常有小范围的代码影响。
- 估计为了充分利用向量硬件，我们如何将“热”循环的数据访问结构化，就像我们对 `vstrip_stable` 和 `hstrip_stabl` 所做的那样。这些修饰经常可以把修改限制到很小的范围内。
- 考虑工作可以在核（明确地说是线程）间传递的可能方式；这关系到如何在代码中分化任务和分配它们的粒度。这可能在数据访问和存储上产生不同的方式，比如我们是如何使用“块”和“排序”的。
- 最终，考虑内存和内存架构的使用和布局。这将会引发一系列代码修改，也会带来与之相称的收获。我们考虑到旋转更新和块方法。

我们提供的最好的通用建议是恰当地应用一个模型，以查看代码对硬件上的运行情况和对其资源的利用情况，恰当应用的编译工具可以大大简化它的实现。写一份最简单的、无法识别硬件的代码，并使用事后自动优化并不会实现高效率。

我们希望我们能够向读者展示这些粗糙的原则如何应用到流行数值计算技术上，尽管当它们应用到其他代码时将会和我们在此展示的有极大不同，但我们相信，在代码编写的过程中考虑它们将会对极大地使得计算受益，同时有望减轻科学家 - 程序员的负担。

2.6 更多信息

- Godunov, S.K., 1959. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik* 89 (3), 271-306.
- Lavallée, P., et al., 2012. Porting and optimizing HYDRO to new platforms and programming paradigms-lessons learn. Tech. rep, Parallel Programming Interfaces, France.
- van Leer, B., 1977. Towards the ultimate conservative difference scheme. IV. A new approach to numerical convection. *J. Comput. Phys.* 23 (3), 276-299.
- Leveque, R.J., 2002. *Finite Volume Methods for Hyperbolic Problems*. Cambridge

University Press ISBN:978-0-521-00924-9.

- Roe, P.L., 1981. Approximate Riemann solvers, parameter vectors, and difference schemes. J. comput. Phys. 43(2), 357-372.
- Satish, M., et al., 2012. Can traditional programming bridge the Ninja performance gap for parallel computing applications? In: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA'12. IEEE Computer Society, Portland, Oregon, pp. 440-451. ISBN: 978-1-4503-1642-2. <http://dl.acm.org/citation.cfm?id=2337159.2337210>.

HBM 上的 SIMD 与并发优化

Jacob Weismann Poulsen^{*}, Per Berg^{*}, Karthik Raman[†]

^{*} 丹麦气象研究所; [†] 美国, Intel 公司

3.1 应用程序: HIROMB-BOOS-MODEL

完整的 HIROME-BOOS-MODEL (HBM) 软件具有双向动态嵌套的特点, 任意数量的嵌套层数能够呈现高分辨率的海洋区域并在狭窄的海峡和航道使用更高的分辨率。除了基本物理组件外, HBM 还拥有可供选择的生物地球化学模块。在支持分布式与共享内存并行化的基础上, HBM 已经发展为一种成熟、高效、便捷、高质量的海洋模型软件。通过修改输入设置标准, HBM 在业务预报及气候等方面的研究工作中得到广泛应用。这种通用模型软件能够使用相同代码处理诸如地理覆盖范围、网格分辨率、嵌套区域数量、模拟时间跨度和一些其他模型特征的不同应用。因为这些应用具有不能被硬编码的动态特征。

运行时系统中用户指定的输入参数、选择使用的具体时间步长和仿真周期数, 是用于区别使用相同源代码执行的业务预报应用和气候情景应用的全部内容。因为动态的输入参数、MPI 任务数和 OpenMP 线程数都是由运行时系统的性能决定的。无论如何选择任务数和线程数, 都必须保证得到相同的二进制运行结果。与可用的其他大多数模型软件相比, HBM 能够在用户选择的串行、MPI 并行、OpenMP 并行或 MPI-OpenMP 混合并行架构上进行移植、编译、执行。HBM 软件已经应用到从气候预测到气候建模等多个重点领域的研究和商业项目中。例如, 丹麦气象研究所 (DMI) 目前正在研发泛欧洲模型, 该模型未来活动的中心包括以下两个主要方面: (i) 在泛欧洲范围内提供相同的预报内容; (ii) 应对泛欧洲和局部区域的气候变化。对于后者, HBM 需要为大气-海洋耦合模型提供可靠的基础服务。解决时间问题是整个项目成功实现的一个重要因素。该泛欧洲模型在狭窄的海峡和临时水域设置了 9 个高分辨率的嵌套区域 (水平方向下降至 185m 以下, 垂直方向下降至 1m)。此外, 该模型拥有了高达 1860 万个活跃湿点, 因此目前的计算需求已经远远超过在 DMI 之前的其他任何单个模型。尽管如此, 从近期的 PRACE 项目 (URL 参见 3.13 节) 中我们可以看出, 现有系统使用 HBM 早期版本进行 5 天预报的时间消耗为 1.5 ~ 2.5 小时。这一结论使得实现业务预测成为可能。然而, 如果要把设置气候时间规模的建模方案在 DMI 中实际应用, 还需要进一步提升模型的并行加速比。

为了使读者更深入地了解有关 HBM 的信息, 可参考 3.13 节。

3.2 关键应用: DMI

该 DMI 提供了许多气象服务, 主要包括天气与气候的预报、预警和监测, 以及大气、陆地、海洋的相关环境影响, 以达到保护丹麦、法罗群岛、格陵兰岛的联邦地区及周边海域、空域人民的生命与财产安全的目的。DMI 的其中一个重要的职责是为诸如暴风雨袭击

丹麦海岸等国家突发事件进行应急准备和积极响应。在 DMI 中，当前版本的海洋环流模型代码已经承担了从 2001 年开始的业务化风暴潮模型并在一些欧盟资助的项目中广泛应用；最新版本的 HBM 项目如今用于提供最近 5 天每天 4 次的水位预报。在丹麦的滨海站该模型能够提供最准确的水位预测。这是通过对比现场观测数据和来自北海及波罗的海等预报中心的预报数据进行不断地验证得到的结果。同一个模型中的三维场景预测的电流和水温通常作为丹麦水域操作油泄露建模的基本数据。在另一种模式设置中，DMI 通过使用源代码完全相同的版本对 MyOcean 项目的波罗的海生产组件进行操作，该项目的关注点是波罗的海中包括物理学和生物地球化学参数在内的海洋状态。该模型设置在不断增加相关区域呈现的分辨率上得到应用并提供了 2.5 天内每天两次的预报。MyOcean 项目的质量被连续地监测和验证。感兴趣的读者可以在本章末尾提到的 MyOcean 网页上进行查阅。

作为一个气象、能源和建设部下属的政府机构，DMI 正在面临着能源预算和限制能耗计算方法的挑战。人们非常希望找到一个对于像 HBM 代码这样的能够有效减少运行时间和能源消耗的方法。我们也正在不断地提高 HBM 的性能，其中本章介绍的工作成果已经运用到 HBM 源代码库中，并作为其中一个重要的组成部分提升了 HBM 整体的运行性能。

3.3 HBM 执行配置文件

作为一个向着 HBM 更高效执行力度发展的准则，实际应用中的 HBM 代码执行时间中不同部分的执行时间所占比重如图 3-1 所示。其中平流示踪部分的时间占到了总时间的 44%，因此它是项目中最主要的部分。实验的结果表明可以通过盐度和温度两种示踪方法得到纯物理运行过程。对于运行像 MyOcean 这样的应用需要额外的 12 个示踪剂表示生物地球化学的变量，平流示踪的运行时间约长 2.5 倍。在任何一种情况下，通过优化平流示踪部分性能来进一步优化 HBM 整体性能的方案似乎非常合理，平流示踪部分优化有以下两种方法：一个是提出一个可实现的新的、更简单、更高效的方案；另一个是尝试提高现有较为复杂方案的性能。由于以下几个原因，第一种优化方案难以实现，因此提高当前方案的性能是优化像 HBM 这样的通用模型代码的唯一出路。

模型	时间所占比重
示踪物平流	44%
湍流模型	16%
动量方程	12%
示踪物扩散	4%
质量方程	2%
其他部分	每个都小于 2% each

图 3-1 HBM 典型应用程序在不同部分代码中的运行时间所占比例。定时程序用于在单节点（未使用 MPI）上运行和单区域模型设置上，对于在较多 MPI 任务和较多嵌套区域上的模型运行情况，各部分程序执行时间百分比会发生变化但是时间开销较大的前 5 个部分间的相互关系将保持不变

对于示踪物平流而言，使用中心差分法或纯迎风格式的相对粗粒度网格划分，可能导致偏差甚至错误的预测结果。总偏差递减（TVD）方案（Harten, 1997）能够预测更陡的峰值并保持不含寄生振荡的单调性（没有“超调”）。但是这些 TVD 的性质往往难以甚至不能证

明通用多维方案的存在,因此应用中的每个坐标方向通常使用一维坐标系保持某种单调性。毫无疑问,任何物理引力模型必须具备质量守恒和单调性。即使是应用在可能发生空间和时间上诸如干燥和湿润骤变的自由表面流动上时,如果能够满足以下两个基本条件(Gross等,2002),的确能够使用一些示踪物平流方法解决。第一个条件是满足一致性和连续性(CWC)概念,即要求离散的示踪物平流方程与离散的自由表面连续方程保持一致;另一个重要条件是在网格单元格表面适当定义变化的高通量。

Kleine(1993)研究并提出了HBM上示踪物平流的实际数值方法。本文作者实现了一种高并发的数值方法,该方法对CWC概念和带有多个示踪物的动态双向嵌套配置的高通量进行了适当修改和调整。此外,作者还调整了所有使用间接寻址方式的数据结构,添加了对MPI和OpenMP的支持,以及使用单指令多数据(SIMD)向量以优化并发性。在BSD授权下,读者可下载、使用本章的特定代码以及与HBM(非完整代码)相关的部分代码。

3.4 HBM 优化综述

为了阐述我们的观点同时简单化不必要的事情,我们将不会处理有关嵌套、气象预报领域及本章提到的其他预报数据等。这些可能会限制其在代码中的使用,但是我们希望能在真实环境的3维数据集中展示我们的研究成果,而不是局限于学术研究中。因此,我们将程序应用在巴芬湾的所有相关数据上,这些数据是从免费的ETOPO2数据集中免费获取的。(URL详见3.13节。)

下一节首先将介绍最重要的且不同于其他环流模型代码的数据结构,然后将描述HBM的线程并行化与SIMD向量化过程。此外,我们详细处理了一些零碎的优化障碍和不明显的优化效果。最后我们结合了多种优化方法并在Intel Xeon处理器和Intel Xeon Phi协处理器上进行性能分析对比。这种特殊应用的实验结果表明,与Intel Xeon处理器相比,我们在一个Intel Xeon Phi协处理器上获得了15%的性能提升。

3.5 数据结构:准确定位位置

作者认为只有在以数据近邻为重点的应用上才能获得基于最新硬件的良好SIMD与线程优化性能。位置,位置,位置……是的,一切都与位置相关,本节将描述整个HBM模型中的面向数据设计方法。

如图3-2所示,我们将3D网格上的点分为活跃计算点(即湿点,陆地上或海平面以下的网格点)和非活跃计算点(在陆地或海平面以下的网格点)。图3-3展示了一个典型的浅水域。我们发现规则3D网格内的活跃计算点数量远少于总计算点数量。例如,图3-3所示的区域中只有11.1%的点是活跃计算点。因此计算模型使用的所有矩阵非常稀疏,这就表明在HBM中应当使用存储量更小、数据更加密集的稀疏结构。我们使用三个数组`msrf(0:,0:)`、`mcol(0:)`和`kh(0:)`分别表示水平面湿点的索引、列中第一个地下点的索引和列的最终长度进行压缩存储。水平面索引数组将水平面网格点 (i,j) 转换为对应的湿点索引。如果 (i,j) 点在陆地或海平面以下则返回0值。地表索引号为`iw`的列索引由`mcol(iw)`函数定义,且`mcol(0)`返回0值。同样,地表索引`iw`的列长度由`kh(iw)`函数定义,且`kh(0)`返回0值。图3-4中的代码展示了3维矩阵`u(:)`上所有湿点的循环过程。我们可以发现`msrf(0:,0:)`从湿点到网格位置 (i,j) 的反向转换过程是由`ind(1:2,:)`确定的。

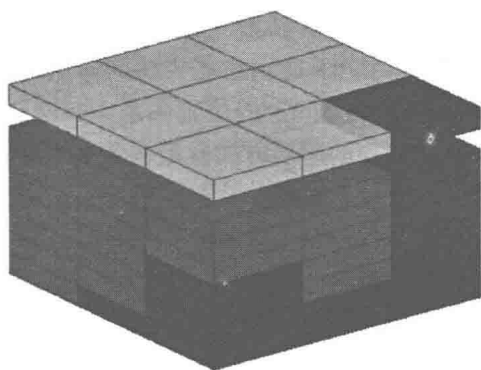


图 3-2 活跃计算点在规则 3 维网格中的分布情况。其中使用浅色代表顶部的活跃计算点分布，中间颜色表示地表下的活跃计算点分布情况，非活跃点用深色表示

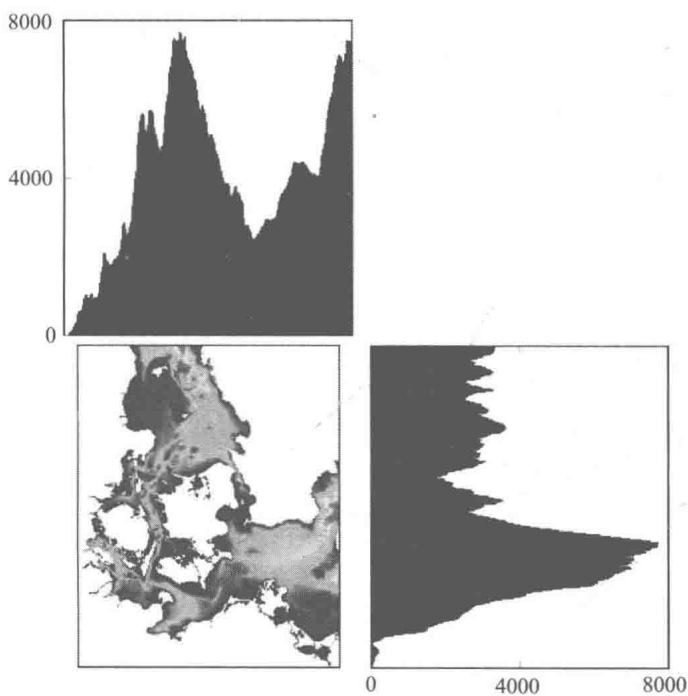


图 3-3 典型不规则计算区域分布图。图中展示了丹麦国内的子水域分布情况，颜色编码表示每个地表点下的湿点数量。其中白色表示陆地，颜色范围从深蓝色表示一个点到暗红色表示 75 个点。右侧的直方图表示沿着每个维度网格线分布的湿点数量。顶部的直方图表示沿着经度网格线分布的湿点数量

图 3-5 展示了我们在 1 维数据结构中划分的三种类型点。该部分代码在水平柱状网格上运行。我们在水平方向上使用间接寻址方式的目的是映射由 $msrf(i, j)$ 得到的列的置换。注意，这里的置换是由一个非结构化的 (i, j) 列来描述的，而不是每列的数据。合理分配数据能够提升应用在表面点和垂直穿过的地下点的直接寻址方式的数据紧密程度。

从图 3-4 的代码段中可以看出，重组数据能够使按列进行独立量化处理的循环操作具有一个完美的跨度减 1 访问模式。但是我们同时也要考虑以下两个方面。首先，由于使用了类似于有限差分的方法，HBM 模型可看作一个 9 点 stencil 计算模型。因此一些循环过程不仅

要处理实际点，还要处理它的东、西、南、北、东北、东南、西北和西南方向上的邻居网格点数据。所以处理列的确定位置 (i, g) 时，还需要访问邻居元素的数据且使用合适的数据分布方式以保持 2 维空间转换至 1 维空间上的位置关系。其次，HBM 模型需要处理整个湿点集合，而不只是一列数据，所以列的遍历方式也是非常重要的一方面。

```

do iw = 1,iw2
  i = ind(1,iw)
  j = ind(2,iw)

  ! Note that iw==msrf(ind(1,iw),ind(2,iw)) by
  ! definition of ind(1:2,:).
  ! All surface wet-points (i,j) reached with stride-1:
  ... u(iw) ...
enddo
do iw = 1,iw2
  kb = kh(iw) ! bottom value at column iw
  if (kb < 2) cycle
  i = ind(1,iw)
  j = ind(2,iw)
  mi0 = mcol(iw) - 2
  do k = 2, kb
    ! all subsurface wet-points (i,j,k) are reached
    ! with stride-1:
    mi = mi0 + k
    ... u(mi) ...
  enddo
enddo

```

图 3-4 活跃湿点循环执行过程的代码段



图 3-5 表示图 3-2 三维网格中的一维数据结构分布情况。其中 $[1:iw2]$ 代表活跃的表面点是， $[iw2+1:iw3]$ 代表活跃的地下点，而不活跃点用 $[0]$ 表示

表面点的任何遍历方式都可以用基本方法实现。事实上，任何一种遍历法最终都能找到最佳临界点。每种遍历方法都会分配一个独特的缓存模式 (D1、L2、L3 和 TLB) 且有些遍历方法要比其他方法好。假设我们能够在三个索引数组数据集上使用大小、延迟、排他性和包容性类型描述目标计算机的存储系统，那么我们将如何调整基本的遍历方式以获得最优的临近点呢？

作为一个说明性例子，我们将展示两种不同的启发式遍历法 H1 和 H2。H1 的遍历方法是从上层的西北角开始，然后从北向南遍历，一旦完成了南方的大部分坐标，然后向东移动一格并再次从北向南计数，以此类推，直至到达东南角的点。在一个真实情形 (巴芬湾的数据集，2 海里范围，分辨率为 77285 表面点和 7136949 地下点) 下，使用这种方法遍历的结果如图 3-6 所示。图 3-6 说明了该数据集在给定索引距离为 $1 \sim 10$ 的空间内邻居的 8 个在地理位置最近的湿点中有多少个能够成为近邻。

基于 9 点 stencil 的计算模型使用启发式 H2 能够提高 D1 利用率。为了找到最佳邻近点，我们让网格单元在索引遍历中尽可能接近邻居单元。如果我们能够在矩形或立方体上保证这些，我们将更进一步完成使用空间填充曲线的任务。在一个如图 3-3 或图 3-8 这样的不

规则数据集上提出一个正确的解决方案非常具有挑战性。

图 3-6 清晰地显示了使用 H2 方法使得 stencil 邻居单元的较大部分（第 1、4 行）在索引空间中距离更近。然而，这种遍历方法是有代价的且索引空间上距离为 10 以外的点与 stencil 中心（第 2、3 行）的距离更大。在 H1 方法中，没有被 D1 命中的邻居点可能会在 L2 或 L3 中命中，而在 H2 方法中它们的距离太远以至于在 TLB 中找不到。这个例子主要用于强调考虑了数量因素：D1、L2、L3 和 TLB 的大小与延迟，以及原始 2 维空间中的距离和 1 维索引空间的距离后，很难制定一个最佳邻近点的方案。

使用H1方法得出的到8个领域的距离

	NW	N	NE	E	SE	S	SW	W
DIST <10	3.32%	100.00%	3.18%	2.00%	3.18%	100.00%	3.32%	2.00%
MAX	400	1	398	399	400	1	398	399
MEAN	282.7	1	281.2	284.8	282.7	1	281.2	284.8
MEDIAN	318	1	318	320	318	1	318	320

使用H2方法得出的到8个领域的距离

	NW	N	NE	E	SE	S	SW	W
DIST <10	64.34%	76.23%	64.34%	83.80%	64.34%	76.23%	64.34%	64.34%
MAX	69326	69313	69326	69327	69326	69313	69326	69327
MEAN	395.1	281.9	432.5	210.9	395.1	281.9	432.5	210.9
MEDIAN	4	3	4	1	4	3	4	1

图 3-6 使用 H1 和 H2 方法得出的到 8 个领域的距离。第 1 行表示 2D 领域中的点也接近索引空间的百分比，其中，接近定义为索引号差值小于 10

我们认为解决最佳邻近点问题是一个公开问题，作为最优解问题将包含所有相关参数并找到最合适的遍历方法。其解决方案必须充分支持任何真实测试用例下模型代码用户可能使用的任何类型缓存系统。然而，为了解决一个优化问题（如求下界的 NP 难题），我们需要使用启发式或近似法。对于本章的测试用例，我们尽可能让用例简单一些并坚持使用启发式 H1 描述，并且有可能为了以后的调整而偏离该问题。

3.6 HBM 上的线程并行

在 HBM 的 MPI 并行化和 OpenMP 并行化中，我们假设所描述的表面点和下面列出的遍历方式是固定的，如果没有这个前提我们将面临着计算复杂度问题。但这一假设规定的简化意味着我们现在面临的不再是计算复杂性。这使我们能够同时使用精确算法和启发式方法分别处理离线与在线的线程或任务的负载均衡问题。此外，它允许我们能够在不同的相关测试用例上评估不同的启发式方法。值得强调的是，不管我们采用何种方法均衡，另一种遍历点方法都将强制处理数据的另一个线程分布。下面是负载均衡问题的定义，图 3-7 展示了点是如何分配到线程中的。

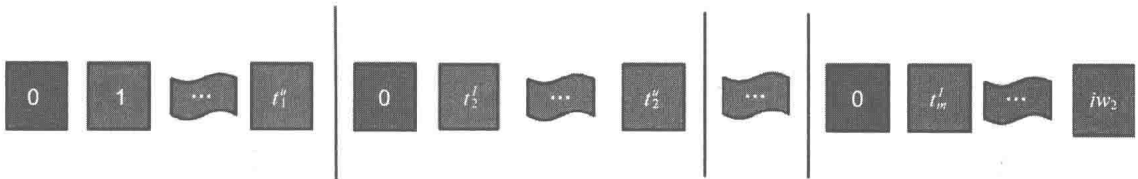


图 3-7 该图显示了每个线程处理一组表面点集合和对应表面点的地下列的子区域。此外，需要注意每个线程还有非活跃点出现

首先令 $I = \{1, \dots, m\}$ 作为列的索引集合, $\{w_1, \dots, w_m\}$ 表示与各列相关的权重, n 表示使用的线程 / 任务数。对于覆盖 I 的不相交的子区间 $I_i = \{[l_i:u_i]\}_{i=1,\dots,n}$ 使得向量 (c_1, \dots, c_n) 的开销为:

$$c_i = \sum_{j=l_i}^{u_i} w_j$$

使用 C_i 的最大值作为隐藏开销的定义。均衡问题就是要找到一个最大限度减少成本的平均点。

数据结构上面的计算复杂性现在已经转变为公认的整数分割问题, 即一个时间复杂度为 $O(nm^2)$ 的精确算法。例如本书中由 Skiena (2008) 编写的 8.5 节。对于中等大小 n 和合理输入数据集的启发式方法是一个典型的贪婪算法, 即将列添加到当前线程中, 直到总和超过每个线程的平均负载数。作为一种选择, 我们必须考虑到这种情况, 这种方法可能会大量重复使用, 并且如果这使总和更接近于每个线程的平均负载, 那么会只把下一列数据添加到当前线程池中。这两种启发式方法的时间复杂度都为 $O(n)$ 且都可以作为在线实验的合理候选方法。这将是 HBM 默认的公平分享版本的替代选择。

巴芬湾的测试用例如图 3-8 所示, 图 3-9 表示的是负载均衡所面临的挑战。

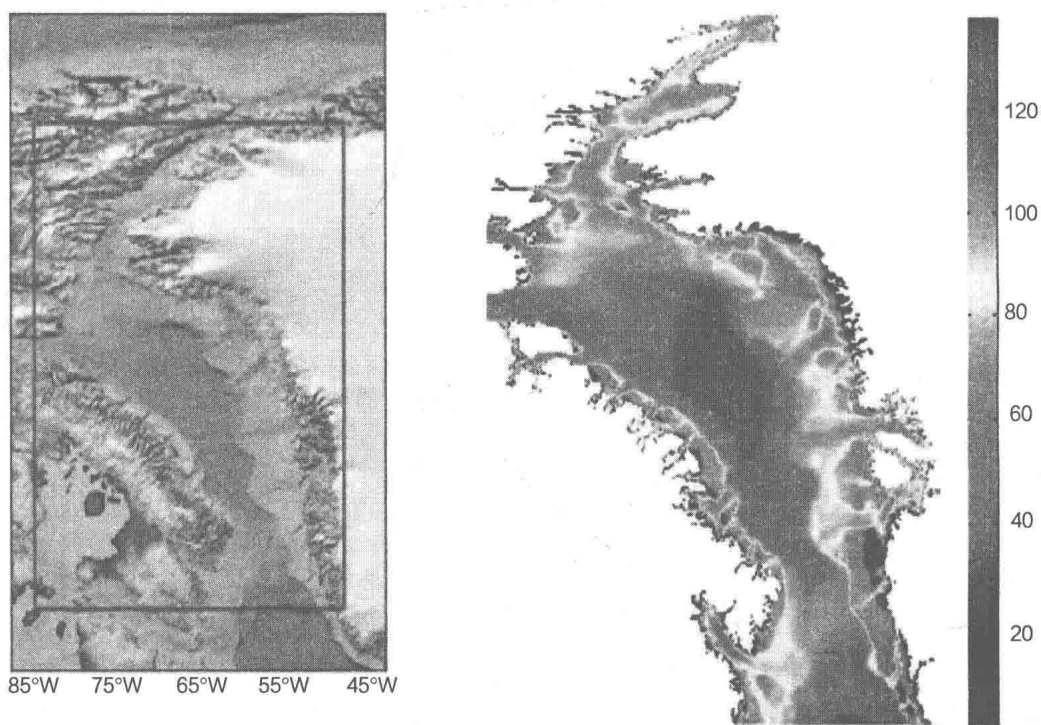


图 3-8 巴芬湾地图测试用例的地图 (左) 和带有平方米深度刻度的相关网格地形的地图 (右)

我们将图 3-9 中的列拆分为 4 个部分, 目的是为了平均列长度与极端列长度的数据。图中的间隔数字表示列的间隔。

我们试图量化工作负载中的默认启发式方法与图 3-10 中对于巴芬湾测试用例使用的精确算法之间的差异性。如图 3-10 所示, 以上述两种方法为核心的解决方案非常相似。我们很难从 2 维或 3 维的纯负载曲线上找到它们的不同之处。不同的曲线只是表示不同的解决方案而已。由 2 维曲线可以看出, 在这种特殊情况下, 使用数据结构思想寻找均衡点的方法可

能比使用拆分成子区间的精确算法更好。我们认为在使用较少线程时没必要使用精确算法。比如在 Intel Xeon 处理器运行只使用了 48 个线程。

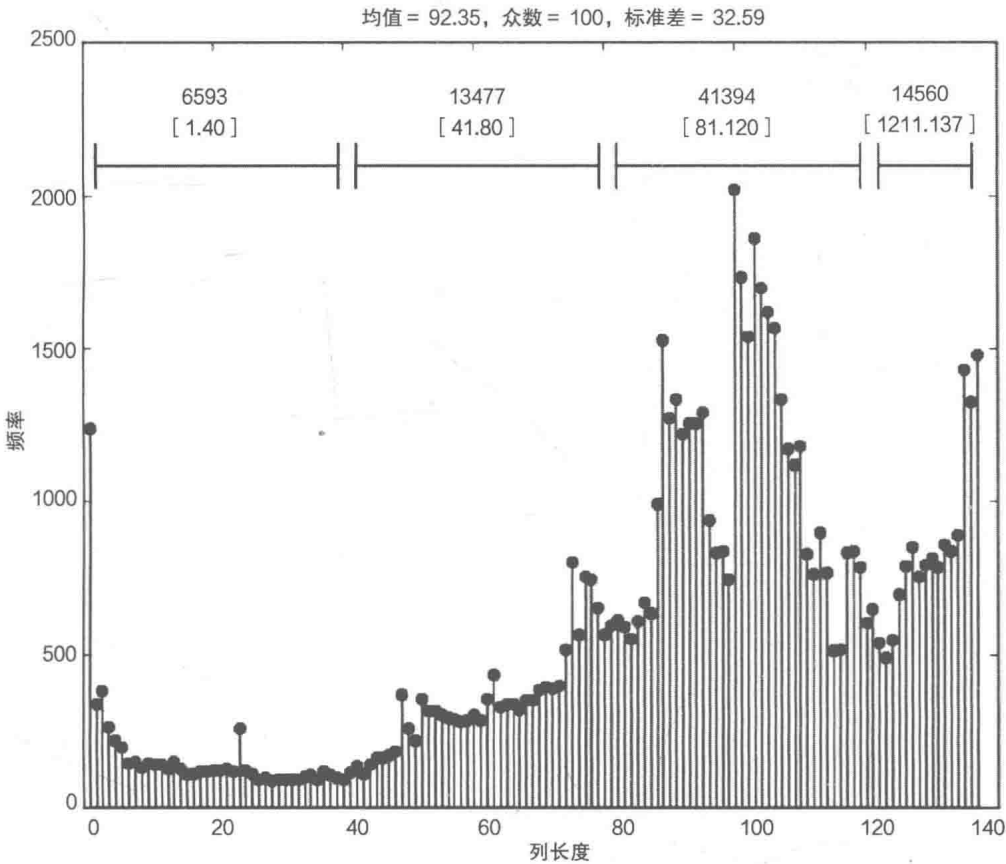


图 3-9 巴芬湾测试用例中的列长度分布

值得注意的是，在定义均衡问题上的权重指的是子权重的总和。同时保留了问题的复杂性。即如果一个点是湿点那么设置权重为 1.0，其他地方为 0.0。图 3-10 使用了这种定义方式。另一种定义也可以使用列的个数，比如，通过 $w = \alpha * s_{wet} + \beta * ss_{wet}$ 公式设置权重，其中， s_{wet} 表示表面湿点的数量， ss_{wet} 表示对应的地下湿点的数量， α 和 β 是可调节参数。

通过进行不同的实验测试，我们试图得到这些权重的相关参数并确定最佳均衡位置系数。更多详细信息请见本章参考文献部分。

一旦我们确定了线程间如何拆分的问题，我们应当考虑应用程序将如何使用线程。一般情况下，我们认为实际应用中的 OpenMP 优化应该与 SPMD 模式方法相似，而不是与基于按列循环的方法一致。因此，OpenMP 并行和 MPI 并行方法之间非常相似，理想情况下，OpenMP 栅栏应该围绕着 MPI halo 交换使得线程间同步时间最短。当然，这就意味着 OpenMP 部分非常大，同时所有循环中所使用的循环与数据结构都是结构化的并统一调用。在 NUMA 架构中，我们为了提高性能需要明确处理“第一次接触”策略。幸运的是，对于我们方法中的所有变量而言，确保正确的 NUMA 布局相对容易。图 3-11 展示了上述部分的代码段。子例程 `domp_get_domain` 将为每个线程提供其上限和下限（分别为 `nu` 和 `nl`），且可以在初始化时读取分解进行赋值或在第一次调用 `kh` 和 `iw2` 特定输入模型时产生。

然后，任何后续调用 `domp_get_domain` 的函数都将在 `idx` 数组中查找并将分解结果 (`nl` 和 `nu`) 传递给当前调用者。这种方法减轻了显式线程作用域的负担。作用域通常在 Fortran 语言中使用，通过创建作用域可以将所有栈变量变成线程的私有变量。

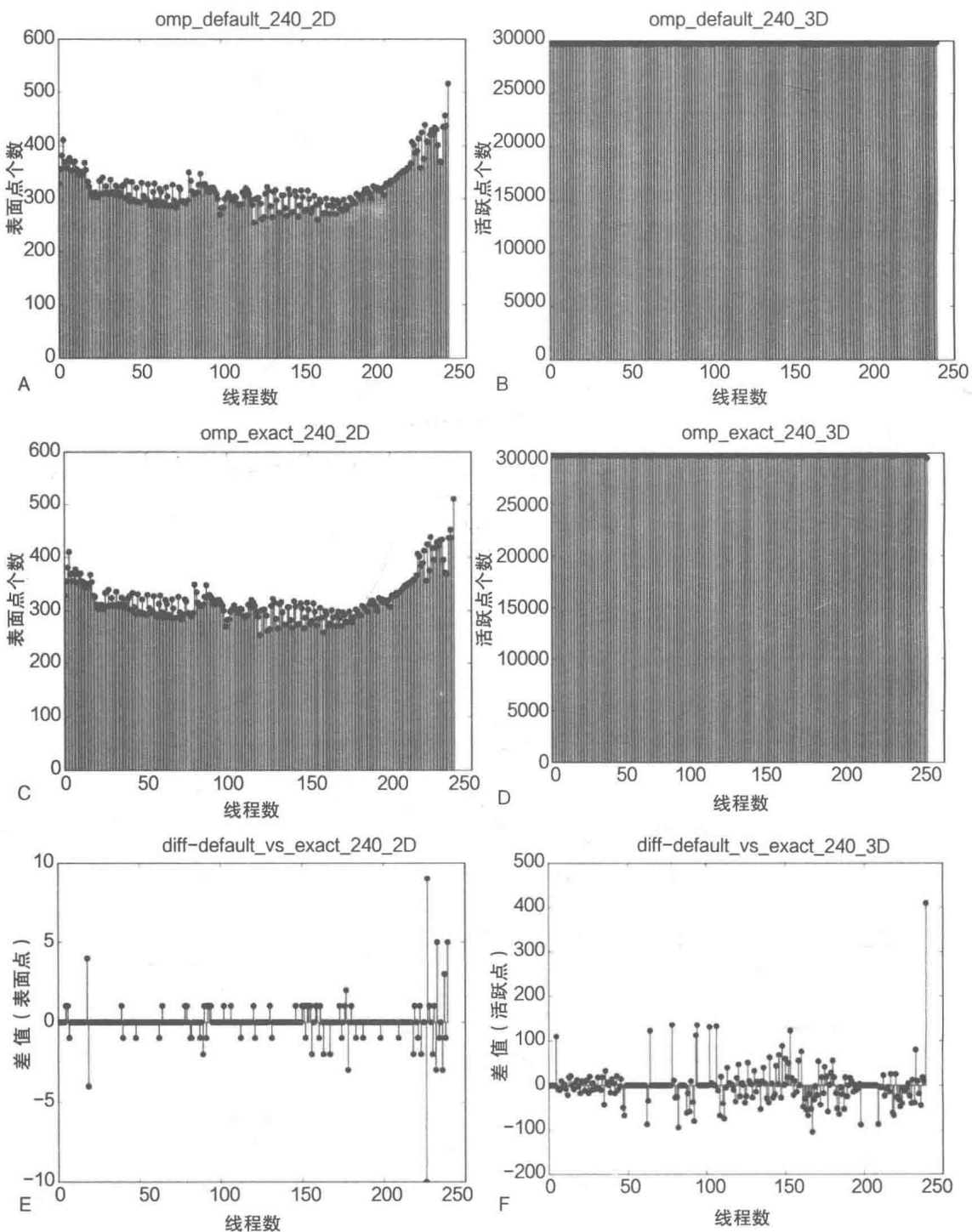


图 3-10 与 240 个线程都相关的表面点和地下点的数量。注意，在 240 个线程上它们之间的差别非常大。所展示的 2 维不平衡可以通过使用权重或者其他遍历列方式进行处理


```

!$OMP PARALLEL DEFAULT(SHARED)
call numa_ft(...); call foo( ... );call bar(...); ...
!$OMP BARRIER
call halo_update(...)
!$OMP BARRIER
call baz( ... );call quux(...); ...
!$OMP END PARALLEL
...
subroutine numa_ft(...)
  call dcomp_get_domain(kh, 1, iw2, nl, nu, idx)
  x(nl:nu) = 0.0_8 ! first touch of thread-local surface
  ...
  do n=nl,nu
    kb = kh(n)
    if (kb > 1) then
      ml = mcol(n)
      mu = ml + kb - 2
      x(ml:mu) = 0.0_8 ! first touch of thread-local subsurface
    endif
  enddo
end subroutine numa_ft

subroutine foo(...)
  call dcomp_get_domain(kh, 1, iw2, nl, nu, idx)
  do iw=nl,nu
    i = ind(1,iw) ! all threadlocal surface wet-points
    j = ind(2,iw) ! (i,j) reached with stride-1
    ... u(iw) ...
  enddo
  do iw=nl,nu
    kb = kh(iw)
    if (kb < 2) cycle
    i = ind(1,iw)
    j = ind(2,iw)
    mi0 = mcol(iw) - 2 ! mid-point in stencil
    mn0 = mcol(msrf(i-1,j)) - 2 ! north-point in stencil
    ub = min(kb,kh(msrf(i-1,j))) ! upper bound on fat k-loop
    do k = 2, ub ! the fat stencil loop
      ! all threadlocal subsurface wet-points (k,i,j) and
      ! (k,i-1,j) are reached with stride-1
      mi = mi0 + k ! mid-point in stencil
      mn = mn0 + k ! north-point in stencil
      ... u(mi) ... u(mn)
    enddo
    do k = ub+1, ... ! remainder loops
    enddo
  enddo
end subroutine foo

```

图 3-11 NUMA “第一次接触” 的代码段

3.7 数据并行：SIMD 向量化

本节将介绍我们在 HBM 模型上做的 SIMD 向量化工作。图 3-12 中代码段展示了通过循环构成整个 HBM 模型的过程。

```

do iw=      ! horizontal - mpi/openmp parallelization
  do k=      ! vertical   - vectorization
    ...
  enddo
enddo

```

图 3-12 HBM 基本循环结构代码段

我们已经实现了最外层循环的 OpenMP 优化，这就意味着循环中将不存在数据依赖。

因此我们也可以使用 SIMD 向量化最外层 iw 循环。然而，对 k 循环做跨度减 1 次访问且同时不在 iw 循环中使用的局限性使得该想法难以实施。现在的 SIMD 硬件在真正意义上还不能支持非同步并行读/写访问。例如，在 VEX 指令集上实现的 `gather` 和 `scatter` 操作可以被视为作用于缓存行上数据元素时的一系列加载/存储的缩写符号。根据 Hofmann 等人 (2014) 的调查，在跨多个缓存行的向量上使用 `gather` 操作的执行效率较低，因此我们在垂直循环上使用普通的向量加载指令会获得不错的效果，因为这里使用的是跨度减 1 模式。

这样做的目的是要确保所有的 k 循环能够很好地进行 SIMD 向量化并且这需要编译器把 k 循环当作跨度减 1 循环来处理。

3.7.1 零散的可优化部分

在处理源代码的过程中，我们发现了一些零散、细小的可优化部分。例如间接引用、假定形状参数和 k 个循环的最内层循环分支。在我们集中精力调整生成 SIMD 代码的同时，我们将列出几个能够转化为性能提升的零碎可优化部分样例。图 3-13 ~ 图 3-15 展示了对初始化需要重写部分的优化伪代码步骤。原始代码中的起始点如图 3-13 所示。

```

real(8), intent(in) :: u(0:) ! OBSTACLE: assumed-shape
...
do iw=
    ! horizontal - mpi/openmp
    ! parallelization
    ...
    i = ind(1,iw)
    jj = ind(2,iw)
    kb = kh(iw)
    do k=2,kb
        ! vertical - vectorization
        mjj = mmk(i,jj,k)
        if (u(mjj) >= 0) then
            ! OBSTACLES: indirect addressing
            ! and branching
            j = jj
        else
            j = jj + 1
        endif
    enddo
    ...
enddo

```

图 3-13 原始代码布局

我们发现原始代码中的间接寻址来自三维索引查询表 `mmk(i,jj,k)`。这个数组将网格点索引数据集 (i, jj, k) 转换到与之对应的湿点索引上，如果 (i, jj, k) 表示的是在陆地上或海平面以下的湿点那么就返回 0 值。如同数据结构中的说明一样，我们可以使用更少的存储和直接寻址方式从 `msrf(i,j)`、`mcol(iw)` 和 `kh(iw)` 数组中获得相同的信息。另外两个障碍是假定形状参数和最内层循环的分支，它们可以通过把代码转换到片段中（见图 3-14）处理（在后面的介绍中我们去掉了额外工作部分，目的是详细描述调优版本所需的关键步骤）。

在伪代码段中，我们使用 `foo(i,j,k)` 作为在网格点 (i,j,k) 上评估更复杂表达的缩写符号；因此 `foo` 并不是代表着数组查找或函数调用。

换言之，点 (i, jj, k) 上的 t_1 是同一列 (i, jj) 上的 t_2 项或从邻居列 $(i, jj+1)$ 到右侧的相似项。注意，虽然图 3-13 中代码段由于使用间接寻址能够一直执行 k 循环的底部，

但如果我们想要在两个主 k 循环中保留纯粹的跨度减 1，那么我们必须要在图 3-14 的代码段中保留一个循环。然后，我们发现把组合中带有两个其他数组 hx 和 u 的 t1 表示为：

```

real(8), intent(in), contiguous :: u(0:)
...
do iw=                                ! horizontal - mpi/openmp parallelization
...
  i = ind(1,iw)
  jj = ind(2,iw)
  mj0 = mcol(iw) - 2
  ub = min(kh(iw),kh(msrf(i,jj+1)))
  j = jj
  mi0 = mcol(msrf(i,j)) - 2
  do k=2,ub                            ! main loop 1 - vectorization
    mjj = mj0 + k                      ! mjj == mi
    mi = mi0 + k
    t1(mjj) = ... * max(sign(one,u(mjj)),0)*foo(i,j,k)
  enddo
  j = jj+1
  mi0 = mcol(msrf(i,j)) - 2
  do k=2,ub                            ! main loop 2 - vectorization
    mjj = mj0 + k                      ! mjj /= mi
    mi = mi0 + k
    t1(mjj) = ... * min(sign(one,u(mjj)),0)*foo(i,j+1,k)
  enddo
  do k=ub+1,kh(iw)                    ! remainder loop
    t1(mjj) = 0
  enddo
enddo

```

图 3-14 第 1 步，初始化部分代码重写

```

do n=n2dl,n2dl
  i = ind(1,n)
  j = ind(2,n)
  mi0 = mcol(n) - 2
  do k=2,kh(n)
    mi = mi0 + k
    t4(mi) = foo(i,j,k)
  enddo
enddo
do n=n2dl,n2dl
  i = ind(1,n)
  j = ind(2,n)
  ub = min(kh(n),kh(msrf(i,jj+1)))
  mi0 = mcol(n) - 2
  me0 = mcol(msrf(i,jj+1)) - 2
  do k=2,ub
    mi = mi0 + k
    me = me0 + k
    uhx = hx(mi)*u(mi)
    t1(mi) = t4(mi)*max(uhx,0) + t4(me)*min(uhx,0)
  enddo
  do k=ub+1,kh(n)
    t1(mi) = 0
  enddo
enddo

```

图 3-15 第 2 步，零碎可优化部分重写

$t1(mjj) * hx(mjj) * u(mjj)$

以至于我们推断出的结果还不如使用系数：

$max/min(hx(mjj) * u(mjj), 0)$

主要原因不在于 if-else 分支的判断语句上，而是所有操作的两次执行上。现在我们的目标就是要消除多余的两次执行过程。首先，我们发现需要同时计算 (i, jj) 和 $(i, jj+1)$ 项并由以下代码做出选择：

```
uhx      = hx(mjj)*u(mjj)
t1(mjj) = foo(i,jj,k)*max(uhx,0)
          + foo(i,jj+1,k)*min(uhx,0)
```

但当程序执行到右侧 $jj+1$ 的邻居列时，我们还要在点 $(i, jj+1, k)$ 上再做一次 `foo` 计算。而此时会得到 `MAX()` 的权重，而不是 `MIN()` 的。

另一方面，我们应该在只运行一次的地方考虑使用“两级火箭”优化。例如，在 (i, j, k) 位置对所有的湿点 (i, j, k) 做 `foo` 计算，在临时数据 (`t4`) 上进行存储，以及最后使用最大或最小函数把它存储到 `t1` 中。图 3-15 展示了该伪代码片段。

上述转换过程对编译器是透明的，并且它将生成 SIMD 优化代码。然而，这些转换不足以生成高效的 SIMD 代码。因此下一步的代码生成和代码轮廓调整工作为进程的优化提供了指导。我们必须确保适当均衡的计算强度和合理的缓存压力。我们通过减少缓存刷新的循环次数数量的方式对后者进行了分析，而对前者做了设计选项的要求。在本章特定的 9 点 stencil 模板计算应用程序中，我们有两种选择。我们可以试图通过将循环拆分为两个子循环的方式最大化向量化长度。这些循环主要是指处理 stencil (如 `iw`) 中点的循环，例如带有两个分支语句的 `iw`，在 8 点邻居上做额外操作的循环 `kh(iw)`。事实上，如果能够还原，邻居循环也可以拆分为 8 个子循环。此外，我们还可以通过构造大的循环结构来同时处理 stencil 的中点和它的 8 个邻居。但要确保跨度减 1 访问的步长数必须是 9 列长度的最小值 `kmin`。然后，需要在长度超过 `kmin` 的列中执行这 8 次循环。通过依次分析循环中的示踪剂平流代码后我们发现，后一种设计 (具有一个大循环和剩下的不超过 8 个循环数) 比拆分为多个 (不超过 9 个) 小循环方法的性能要好许多。

3.7.2 过早抽象是万恶之源

实际上，HBM 的示踪物平流部分在另一个示踪物的数量的垂直循环意义上有些特殊。这个最内层 `nc` 循环只存在于 HBM 代码的示踪物相关部分。垂直循环内部的示踪物循环代码如图 3-16 所示。

```
do iw=      ! horizontal - mpi/openmp parallelization
  do k=      ! vertical   - vectorization
    ...
    do nc=   ! innermost loop (in advection) with
              ! number of tracers
      ...
    enddo
  enddo
enddo
```

图 3-16 带有示踪物循环的基本循环结构

本节将介绍最简单的硬件抽象 (2 维数组) 是如何导致在 Intel Xeon Phi 协处理器上 2

倍性能损失的。这从侧面验证了我们之前的假设——硬件抽象代价太大。

最初的设想是保持 2 维数组中所有的示踪部分以及使用一种相似的方式连贯地处理该部分。图 3-17 展示了简化的代码段。

```
1 do k=2,kmax
2   k1 = k+off1
3   k2 = k+off2
4   t(1:nc,k) = t(1:nc,k) + A(k)*(B(1:nc,k1)-B(1:nc,k2))
5 enddo
```

图 3-17 简化的代码段

通过编译这部分代码我们发现：

编译器使用动态 nc 向量化 nc 循环：

```
(4): (col. 7) remark: LOOP WAS VECTORIZED
```

编译器使用静态 nc 向量化 k 循环：

```
(1): (col 7) remark: LOOP WAS VECTORIZED
```

生成的代码如下：

AVX (基本上是一个软件集合操作)：

```
...
vmovsd (%r10,%rcx,2), %xmm6
vmovhpd 16(%r10,%rcx,2), %xmm6, %xmm6
vmovsd 32(%r10,%rcx,2), %xmm7
vmovhpd 48(%r10,%rcx,2), %xmm7, %xmm7
vinserftf128 $1, %xmm7, %ymm6, %ymm7
...
```

MIC (一个硬件集合)：

```
vgatherdpd (%r13,%zmm2,8), %zmm6{%k5}
...
```

我们实现了我们所希望达到的，既不是 MIC 目标也不是 AVX (IVB) 目标。必须承认的是，SNB/IVB 架构上的 256 位非对齐加载 / 存储是已知问题，所以软件集合不像它看起来的那么差。下面将分析我们没有得到普通向量负载的原因。静态 nc 意味着要进行展开，展开又意味着优化者把循环看作跨度减 nc 循环。nc=2 的 c.f. 代码段如图 3-18 所示。

```
do k=1,kmax
  k1 = k+off1
  k2 = k+off2
  t(1,k) = t(1,k) + A(k)*(B(1,k1)-B(1,k2))
  t(2,k) = t(2,k) + A(k)*(B(2,k1)-B(2,k2))
enddo
```

图 3-18 展开示踪物循环

我们知道编译器能够做到这一点，但是我们还是建议使用图 3-19 中的样式。

2维与1维（通过nc=2载入）混合的处理过程							
zmm1	=	t(1,1)	t(2,1)	t(1,2)	t(2,2)	t(1,3)	t(2,3)
zmm2	=	t(1,5)	t(2,5)	t(1,6)	t(2,6)	t(1,7)	t(2,7)
zmm3	=	B(1,1+k1)	B(2,1+k1)	B(1,2+k1)	B(2,2+k1)	B(1,3+k1)	B(2,3+k1)
zmm4	=	B(1,5+k1)	B(2,5+k1)	B(1,6+k1)	B(2,6+k1)	B(1,7+k1)	B(2,7+k1)
zmm5	=	B(1,1+k2)	B(2,1+k2)	B(1,2+k2)	B(2,2+k2)	B(1,3+k2)	B(2,3+k2)
zmm6	=	B(1,5+k2)	B(2,5+k2)	B(1,6+k2)	B(2,6+k2)	B(1,7+k2)	B(2,7+k2)
zmm7	=	A(1)	A(1)	A(2)	A(2)	A(3)	A(3)
zmm8	=	A(5)	A(5)	A(6)	A(6)	A(7)	A(7)
2维与1维（算法层面）混合的处理过程							
zmm9	=	zmm1	+	zmm7	*	(zmm3 - zmm5)	!k=1, 4; nc=1, 2
zmm10	=	zmm2	+	zmm8	*	(zmm4 - zmm6)	!k=5, 8; nc=1, 2

图 3-19 混合 2 维和 1 维数据的适当处理

```
1 do k=2, kmax
2   k1 = k+off1
3   k2 = k+off2
4   t1(k) = t1(k) + A(k) * (B1(k1)-B1(k2))
5   t2(k) = t2(k) + A(k) * (B2(k1)-B2(k2))
6 enddo
```

图 3-20 只使用 1 维数组的编译器重写

因此，要优化生成的代码我们必须简化并更改代码以便也使用 1 维数组表示示踪物，即，图 3-20 所示的使用 t1(1:) 和 t2(:)，而不是 t(1:2,:)。

此外，我们可以尝试互换循环（用 nc 倍的存储带宽和缩短 1/nc 的向量长度）。这里总的问题是含有多个混合维度变量的循环。如果所有的变量具有相同的维度，那么编译器将分解循环并生成正确的代码。

3.8 结果

本节将展示 HBM 的性能和平流模块的一些特点。正如前面提到的一样，我们选择调整对流模块是因为这部分的执行时间在单节点的 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上都占到了总时间的 40% 左右。协处理器上消耗的时间是优化这段代码前在处理器上时间的 3 倍。在数据结构采用上述优化后，我们对巴芬湾测试用例的 HBM 平流模块做了对比分析，其 SIMD 和纯 OpenMP 并行化在协处理的本地模式运行时性能比双插槽处理器上的性能提高了约 15%。然而，我们的优化工作并未就此停止，我们发现一些提高缓存利用率的新方法，这将进一步提高并行性能。

所有的巴芬湾测试用例都是不规则的且把这些不规则的网格分割成一个负载均衡问题比理想的立方体测试用例更难处理。性能、上述数据局部性、SIMD 和当前优化方法都极大提高了 Xeon (E5-2697 v2) 的性能并得到比原始 Intel Xeon Phi 协处理器 (KNC) 更好的性能。对于巴芬湾测试用例使用纯 OpenMP 并行化的方法在 KNC 7120A 上的并行性能比在处理器 (E5-2697 v2) 上提高了 15% 左右，如图 3-21 所示。

存储器带宽测量结果表明应用程序在协处理器上达到了约 90% 的实际可实现带 (POP)，并且在处理器上达到的实际带宽为 100%。通过 Stream Triad 方法测量的实际带宽峰值比处

理器的快 2.15 倍。处理器也使用了 1.45 倍的功率并达到了它的峰值。我们对于协处理器的功耗测量只测量了协处理器专用卡电源。因此从带宽数据我们可以推断，Intel Xeon Phi 协处理器在数据重用方面还有提升空间，这将进一步充分利用各部分时间并尽可能达到 100% 的带宽峰值利用率。

当读者读到本章的 SIMD 优化部分时会发现高效利用向量化对于提升性能（尤其是在 Intel Xeon Phi 协处理器上）而言是一种重要的手段。向量化强度（VI）是向量化后的循环执行情况和使用 VPU 的效率的评价方法。由于 512 位向量长度限制 VI 值在单精度代码中不能超过 16 且在双精度代码中不能超过 8。如果它的值远小于上述的规定值，这就意味着向量化循环效果非常差或 VPU 没有高效使用（访问次数低，且由于存在条件语句和向量化，屏蔽指令难以使用）。平流模块代码使用双精度浮点数据类型，因此 VI 值约为 7 表明了向量化循环非常好且在协处理器上的 VPU 利用率达到了 82% ~ 96%。

输入	Xeon E5-2697 v2 时间 (s)	KNC 7120A 时间 (s)	Xeon BW (GB/s)	POP Xeon (%)	KNC BW (GB/s)	POP KNC (%)	KNC VI	P/W
BaffinBay_2nm	81.581	70.662	89.23	100	161.55	89.75	7.04	1.63倍
BaffinBay_1nm	320.81	273.095	88.56	100	160.69	89.27	6.87	1.62倍

图 3-21 平流模块性能总结

性能 / 功率（PW）的提升是指不同测试用例在基本的处理器（Xeon E5-2697 v2）和协处理器（KNC 7120A）得到解决方案的时间比乘以 1.45 系数后的结果。我们使用 1.45 作为系数是因为处理器（Xeon E5-2697 v2）达到峰值带宽消耗的功率是协处理器（KNC 7120A）功耗的 1.45 倍，另外当前应用已经获得了 90% 的峰值带宽。

3.9 详情分析

我们使用 Intel VTune Amplifier XE 2013 工具对结果进行了分析。虽然我们在优化过程中使用了许多性能度量方法，但我们在图 3-22 中只列出最相关的部分。

度量	使用的硬件事件
带宽	UNC_F_CH0_NORMAL_READ、UNC_F_CH1_NORMAL_READ、 UNC_F_CH0_NORMAL_WRITE、UNC_F_CH1_NORMAL_WRITE
向量强度	VPU_INSTRUCTIONS_EXECUTED、VPU_ELEMENTS_ACTIVE

图 3-22 用于计算带宽和向量效率的 PMU 事件

VPU_INSTRUCTIONS_EXECUTED 参数负责统计所有向量指令，且 VPU_ELEMENTS_ACTIVE 参数负责统计针对向量操作（内存和算法）的所有 VPU 可用有效通道。

对于所使用的配置文件，Intel VTune Amplifier 工具的硬件事件采样收集器能够通过配置文件概述使用性能监控单元（PMU）计数器溢出功能的应用程序。使用上述测量值的用例数据大约有 1000 万条。而且采样不能保证数据的 100% 准确。基于事件采样的平均开销大约是每 1 毫秒的 2% 采样间隔。

3.10 处理器与协处理器可扩展性对比

图 3-23 展示了 BaffinBay_2nm 测试用例在 Intel Xeon Phi 协处理器 (KNC) 上的 OpenMP 可扩展性。其中 x 轴表示内核数目, y 轴表示转化为 1.0 的吞吐量 ($1/\text{时间}$)。图中时间的单位是秒。上述内容所使用的内核线程数都是 4 个。从图中可以发现我们的平流模块在 KNC 上获得了非常好的内核可扩展性。理想化的内核并发性和可扩展性对与 Intel Xeon Phi 协处理器非常重要。

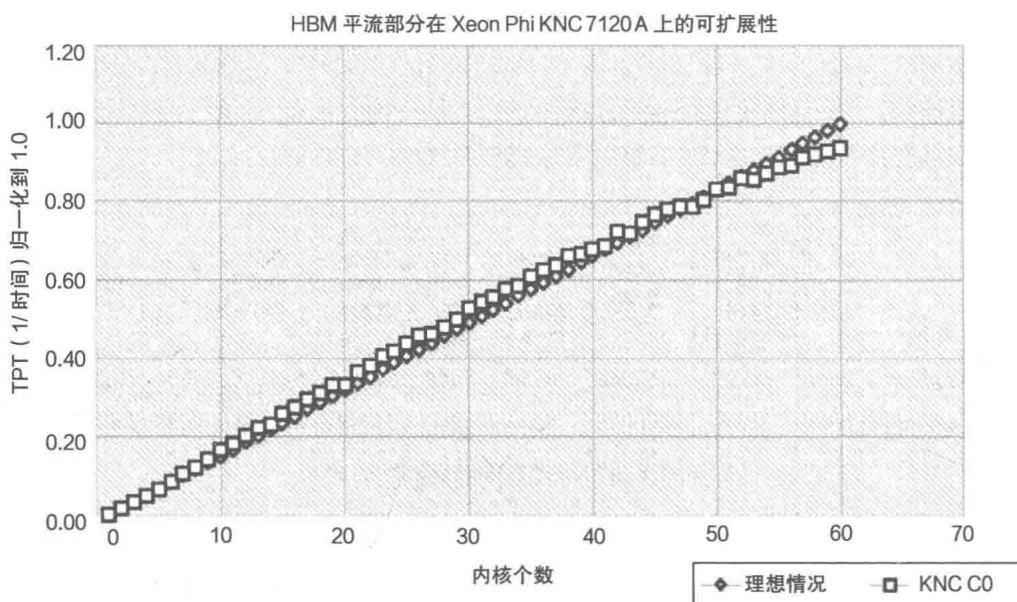


图 3-23 Intel Xeon Phi 协处理器 (KNC 7120A) 的 OpenMP 线程可扩展性

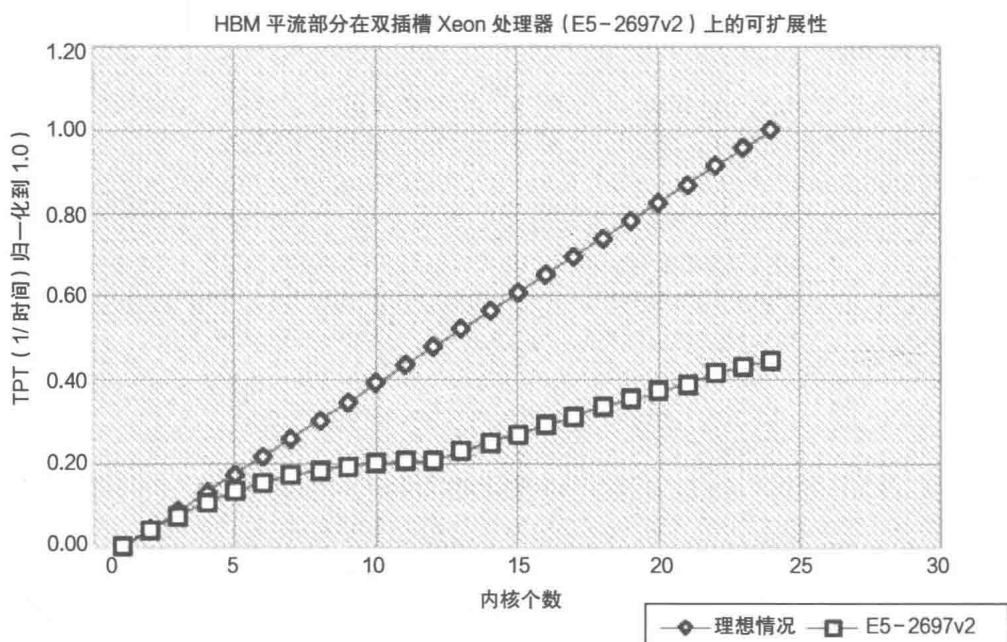


图 3-24 双插槽 Xeon 处理器 (E5-2697 v2) 上 OpenMP 线程的可扩展性

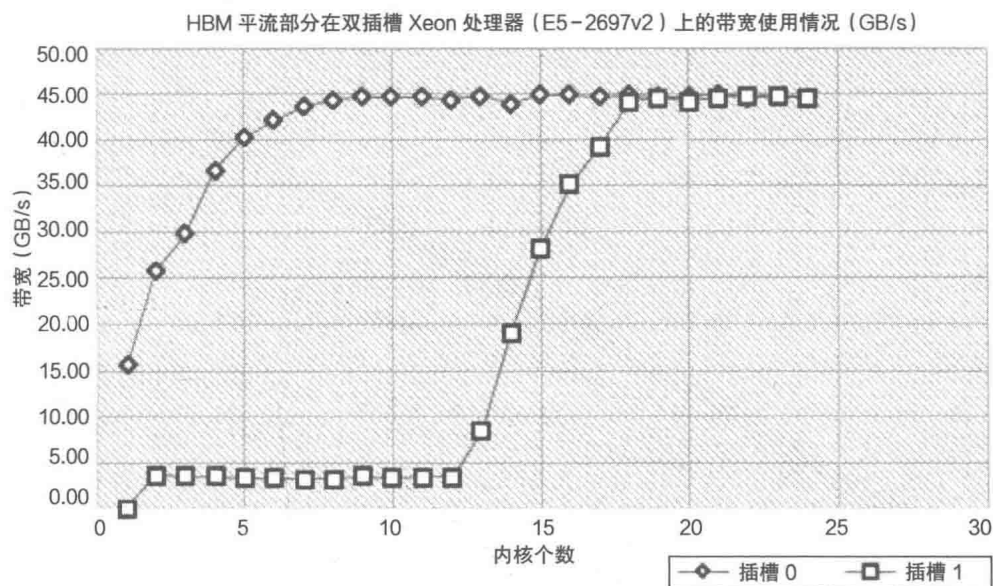


图 3-25 双插槽 Intel Xeon 处理器上内核的带宽可扩展性

双插槽 Intel Xeon 处理器上的可扩展性测试结果如图 3-24 所示。由图可知，其可扩展性不是非常好。主要原因是受到了内存带宽的限制。我们为图 3-24 中各插槽的内核使用情况绘制了内存带宽利用率图，如图 3-25 所示。一个 Xeon E5-2697 v2 处理器上每个插槽的内存带宽峰值大约为 43GB/s。因此，双插槽的内存带宽峰值应该在大约 84 ~ 88GB/s 的范围内。我们测试的应用程序好像达到了约 6 核 12 线程 0 号插槽的内存带宽峰值，通过分析图 3-24 的扩展性图可以得出，在峰值点后的可扩展性开始变得平缓了。然后在第二个插槽（1 号插槽）13 核上，我们发现扩展性有所增长，但是在 17 核上再次达到 1 号插槽的内存带宽峰值，从而限制了整个双插槽系统的性能提升。综上所述，我们可以得出这样的结论，由于内存带宽的限制，处理器性能和可扩展性的提升是有限的，但是 Intel Xeon Phi 协处理器自身的特点使得这种应用程序不因内存带宽的限制而影响提升性能。

图 3-26 展示了协处理器重要的并行性能和良好的并发性。x 轴表示从 1 到 60 的协处理数目，y 轴表示平流模块的吞吐量（1/时间）。Intel Xeon Phi 协处理器上平流模块代码的性能似乎和 100% 并行（没有串行代码）的 Amdahl 图相匹配。例如，如果应用程序中有 1%、3% 和 5% 的串行代码，那么问题求解时间将会显著增加，因此吞吐量将会明显减少，如图 3-26 所示。这完全符合 Amdahl 定律。

3.11 CONTIGUOUS 属性

FORTRAN 2008 中的 CONTIGUOUS 属性明确规定了假定形状的数组必须是连续的或指针只能指向连续的对象。在很多情况下即使不指定 FORTRAN 的标准规格，对象也可以是 CONTIGUOUS 事件。但这种 CONTIGUOUS 属性使得编译器优化变得更容易，这些优化依靠占用连续内存块的对象内存布局。

图 3-27 所示数据清楚地表明 CONTIGUOUS 属性提升了 Intel Xeon (E5-2697 v2) 大约 1% ~ 2% 的性能，对 Intel Xeon Phi 协处理器 (KNC 7120A) 的性能提升更加明显，达到了大约 20%。

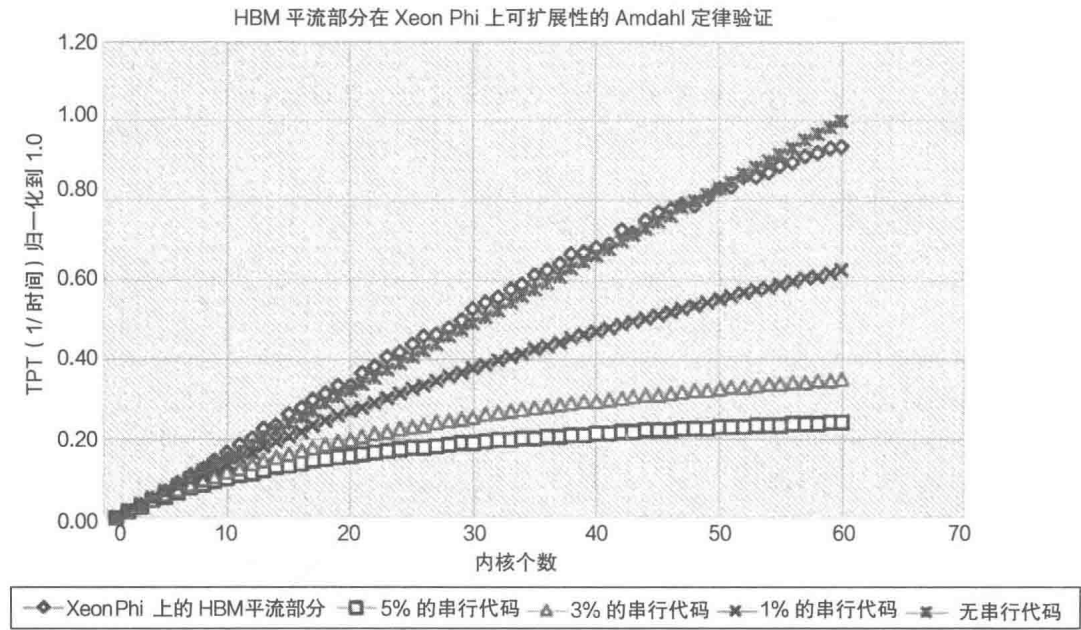


图 3-26 Intel Xeon Phi 协处理器上串行代码对 Amdahl 定律的验证和影响

	具有“CONTIGUOUS”属性		不具有“CONTIGUOUS”属性		性能差异	
输入	双插槽 Xeon 时间 (s)	KNC 7120A 时间 (s)	双插槽 Xeon 时间 (s)	KNC 7120A 时间 (s)	双插槽 Xeon 加快的时间	KNC 7120A 加快的时间
BaffinBay_2nm	81.165	70.653	83.283	85.490	1.02倍	1.21倍
BaffinBay_1nm	320.97	273.158	323.401	333.252	1.01倍	1.22倍

图 3-27 在 Xeon 和 Xeon Phi 上使用 Fortran 的 CONTIGUOUS 属性获得的性能提升

3.12 总结

本章中使用的许多技术和优化思路具有广泛的适用性：对于数据局部性问题可以使用线程和向量化技术。本章中所采用的思维方式可供设计高性能应用程序的程序员借鉴。这样优化后的程序才能在 Intel Xeon 处理器和 Intel Xeon Phi 处理器上都获得性能提升。

我们认为只有专注于数据邻近性的实现才能获得硬件模型上良好的 SIMD 和线程性能。在 HBM 模型中一直使用面向数据的设计能获得较好的性能。如果要选择符合并行的大气环流模式代码，数据结构的设计是最基础也是最关键的部分。在使用 SIMD 向量化代码的过程中不要忽略了克服琐碎的和一些不那么明显的障碍。技术虽然都可以在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上使用，但是协处理器的总内存带宽更大能够使程序获得更好的扩展性。对于本章的特殊应用，相对于在单节点 Intel Xeon Phi 协处理器上，在 Intel Xeon 处理器上获得了 15% 的性能加速比。Intel Xeon Phi 协处理器上的功耗效率是非常吸引人的。虽然协处理器卡的内存限制现在是获取更好性能的瓶颈，但是我们相信在下一代的 Xeon Phi 协处理器（Knights Landing）上这一问题会得到明显改善。

3.13 参考文献

Gross, E.S., Rosatti, G., Bonaventura, L., 2002. Consistency with continuity in conservative advection schemes for free-surface models. *Int. J. Numer. Methods Fluids* 38 (4), 307–327.

- Harten, A., 1997. High resolution schemes for hyperbolic conservation laws. *J. Comput. Phys.* 135 (2), 259–278. With an introduction by Peter Lax, Commemoration of the 30th anniversary of *J. Comput. Phys.*
- Hofmann, J., Treibig, J., Hager, G., Wellein, G., Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips, the Workshop on Programming Models for SIMD/Vector Processing at PPOPP 2014, Orlando, FL, Feb 16, 2014. Preprint: arXiv:1401.7494.
- Kleine, E., 1993. Die konzeption eines numerischen verfahrens für die advektionsgleichung. Technical Report. Bundesamt für Seeschifffahrt und Hydrographie, Hamburg.
- Skiena, S., 2008. *The Algorithm Design Manual*, second ed. Springer, London.
- Xu, P., Tirthapura, S., 2012. A lower bound on proximity preservation by space filling curves. In: *IPDPS*, pp. 1295–1305.

3.14 更多信息

- 有兴趣的读者可以参考 Berg and Poulsen (2012) 和 Poulsen and Berg (2012a) 了解关于 HBM 代码的实施细则、设计目标和代码维护与测试标准等更多资料。这两个报告中还给出了不同应用程序的性能研究情况，第三个报告 (Poulsen and Berg, 2012b) 提供了线程可扩展性研究信息。
- Berg, P., Poulsen, J.W., 2012. Implementation Details for HBM. DMI Technical Report No.12-11. Technical Report, DMI, Copenhagen. <http://beta.dmi.dk/fileadmin/Rapporter/TR/tr12-11.pdf>.
- Poulsen, J.W., Berg, P., 2012a. More Details on HBM—General Modelling Theory and Survey of Recent Studies. DMI Technical Report No. 12-16. Technical Report, DMI, Copenhagen. <http://beta.dmi.dk/fileadmin/Rapporter/TR/tr12-16.pdf>.
- Poulsen, J.W., Berg, P., 2012b. Thread Scaling with HBM. DMI Technical Report No. 12-20. Technical Report, DMI, Copenhagen. www.dmi.dk/fileadmin/user_upload/Rapporter/tr12-20.pdf.
- She, J., Poulsen, J.W., Berg, P., Jonasson, L., 2013. Next generation pan-European coupled Climate-Ocean Model—Phase 1 (ECOM-I). PRACE Final Report.
- HIROMB: High Resolution Operational Model for the Baltic Sea, <http://www.hiromb.org/>.
- BOOS: Baltic Operational Oceanographic System, <http://www.boos.org/>.
- MyOcean 是专用于 GMES (全球环境与安全监测) 海事服务海洋监测、预报实施的主要欧洲项目, <http://www.myocean.eu/>.
- PRACE: Partnership for Advanced Computing in Europe, <http://www.prace-ri.eu/>.
- ETOPO1 集成了土地地形和海洋测深的精度为 1% 的全球地形模型, <http://www.ngdc.noaa.gov/mgg/global/global.html>.
- 从 <http://lotsofcores.com> 上下载本章和其他部分章节代码。
- 对空间填充曲线 (SFC) 感兴趣的读者可能对以下内容感兴趣:
 - Mitchison, G., Durbin, R., 1986. Optimal numberings of an $N \times N$ array. *SIAM J. Algebr. Discrete Meth.* 7 (4), 571–582.
 - Niedermeier, R., Reinhardt, K., Sanders, P., 1997. Towards optimal locality in mesh-indexings. In: *Proceedings of the 11th International Symposium on Fundamentals of Computation Theory (FCT'97)*, Number 1279 in LNCS, Springer, pp. 364–375.
 - Niedermeier, R., Reinhardt, K., Sanders, P., 2002. Towards optimal locality in mesh-indexings. *Discrete Appl. Math.* 117 (1–3), 211–237.

流体动力学方程优化

Antonio Valles^{*}, Weiqun Zhang[†]

^{*} 美国, Intel 公司; [†] 美国, 劳伦斯伯克利国家实验室

随着 Intel Xeon 处理器和 Intel Xeon Phi 协处理器在各节点上的内核数和线程数的不断增加, 基于节点层上的混合编程应用 (MPI+OpenMP) 出现了新的优化方法。然而, 现在的多数应用程序不适合在单节点上采用多线程优化性能。本章介绍由美国劳伦斯伯克利国家实验室 (LBNL) 和 Intel 公司合作完成的 SMC 应用程序的并行性能分析和优化方法。

4.1 开始

SMC 是一种求解多组反应可压缩的流体动力学 Navier-Stokes 方程的并行算法。它是由 LBNL 的计算机科学与工程中心 (CCSE) 开发的。简化版的 SMC 代理应用程序被高性能计算系统厂商和美国能源部湍流模型的百亿亿次模拟中心用于系统模拟和编程模型分析, SMC 的燃烧模拟结果如图 4-1 所示。本章对简化版的 SMC 进行分析和优化。

主要采用 Fortran 语言和部分 C 语言编写 SMC 应用程序。它是由 CCSE 开发的基于 BoxLib 软件架构实现的 MPI-OpenMP 混合同行方法。SMC 采用有限差分法来处理在三维坐标中用笛卡儿网格点来表示的多组流体动力学方程的数值解。使用 Fortran 的四维数组来分别储存坐标 x 、 y 、 z 和组件的数据。由于在 Fortran 中的数组是按列优先存储的, 因此数据在内存中沿 x 轴方向是连续的。每个纬度上空间导数第 8 阶 stencil 使用的数据来自该维度上的 9 个网格点。

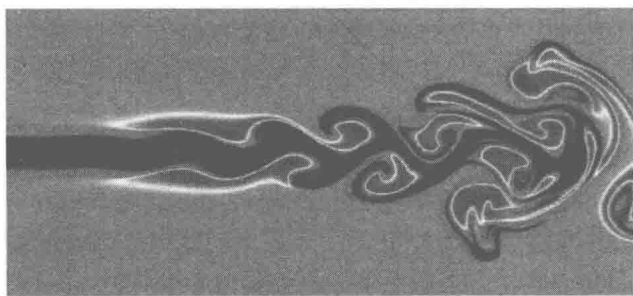


图 4-1 使用化学机理的 39 种二甲醚燃烧模拟的温度部分情况

本章介绍的 SMC 版本都可以在 GitHub 上下载 (见 4.9 节)。

本章将介绍 SMC 的 4 个主要优化方法。下面 5 个版本都已在 GitHub 上发布。本章详细讨论的内容如下。

- 1) 1.0 版本: 基础版本 (Baseline)。
- 2) 2.0 版本: 线程盒 (Threadbox)。
- 3) 3.0 版本: 栈内存 (Stack memory)。

- 4) 4.0 版本：分块 (Blocking)。
- 5) 5.0 版本：向量化 (Vectorization)。

GitHub 上的 README 文件说明了如何编译和运行该程序。该程序提供了许多编译选项 (MPI、OpenMP、GNU 编译器、Intel 编译器等)。花点时间研究一下 `inputs_SMC` 文件可以学习不同的编译选项和运行选项。

程序的性能由推进 5 个时间步长的时间来衡量，其中不包含初始化计算模型的时间。在实际运行时间中也会去掉初始化时间，因为初始化在程序执行过程中只运行一次，而实际运行时间将持续数天甚至数周。

SMC 是一个能够在多个 MPI 进程和 OpenMP 线程上运行的混合作业负载。但是，由于 SMC 已经证明了在 MPI 上较好的弱可扩展性，使用高达 16000 个 MPI 进程能够达到不低于 90% 的效率。因此在 SMC 上 MPI 不是性能瓶颈。接下来将重点讨论如何使用 OpenMP 和向量化方法有效提升并行性。我们在 1 个节点的 1 个 MPI 进程上只使用一个模拟盒子，该模拟盒包含了 $128 \times 128 \times 128$ 个网格点。

在以下两个不同测试平台进行对比分析。

- 1) Intel Xeon Phi 协处理器 7120P (代号为 Knight Corner):
 - 61 核处理器，每个内核有 4 个线程，共 $61 \times 4 = 244$ 个线程。
- 2) Intel Xeon 处理器 E5-2697 v2 (代号为 IvyBridge-EP):
 - 两个插槽，每个有 12 个内核，这些内核支持 Intel 超线程技术，共 2 个插槽 \times 每个插槽 12 核 \times 每个内核 2 个线程 = 48 个线程。

4.2 1.0 版本：基础版本

我们首先展示了使用 SMC 1.0 版本的基本可扩展性测试结果。图 4-2 展示了协处理器 OpenMP 线程可扩展性，图 4-3 展示了处理器 OpenMP 的线程可扩展性。图中展示了 SMC 推进时间的倒数和 OpenMP 线程数之间的关系。图上有两个 y 轴坐标：左边的 y 轴坐标是时间的倒数，右边的 y 轴坐标是单位线程数的性能加速比。我们用左边的 y 轴坐标展示平台和版本 (越高越好)，我们用右边的 y 轴坐标来了解它的单个线程运行情况及其可扩展性。

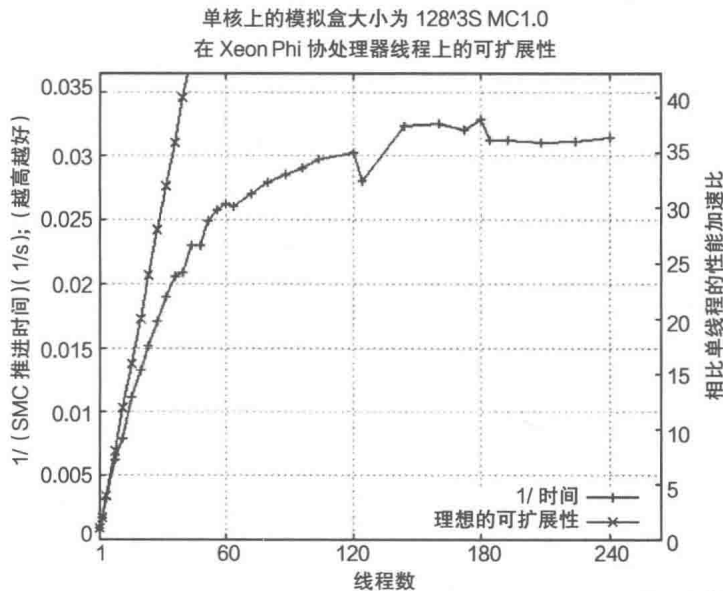


图 4-2 SMC 1.0 版本：协处理器线程可扩展性

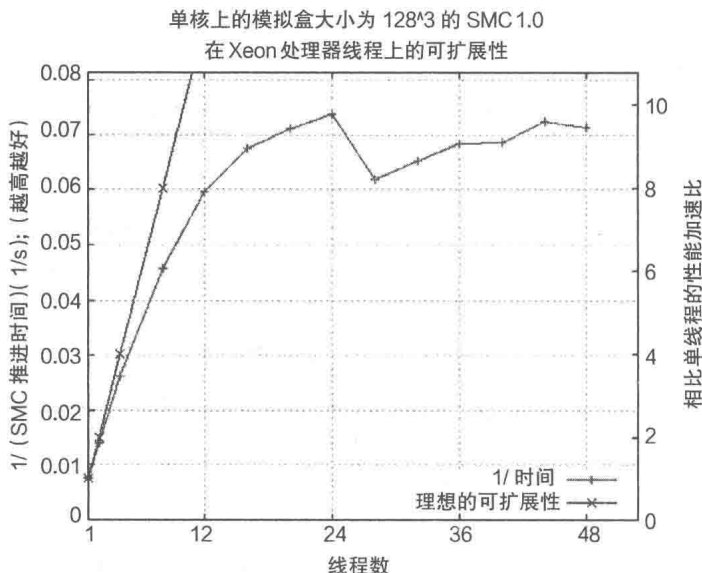


图 4-3 SMC 1.0 版本: 处理器线程可扩展性

从上述两张图我们可以看出, SMC 1.0 版本的可扩展性并不好。我们使用了一个具有 240 个 OpenMP 线程的协处理器, 但仅得到在 Intel Xeon Phi 协处理器单线程上运行时约 38 倍的性能加速比。同样, 我们在一个拥有 48 个 OpenMP 线程的处理器上运行只得到了在 Intel Xeon Phi 处理器单线程上运行时约 10 倍的性能加速比。

我们使用 Intel VTune Amplifier XE 工具对 OpenMP 出现的性能问题和潜在可优化部分进行了分析。图 4-4 展示了使用 Intel VTune Amplifier XE 2015 版本对 SMC 1.0 在 Intel Xeon Phi 协处理器上进行 OpenMP 性能分析的三个截图。图中的第一部分是 OpenMP Analysis, 其结果表明了该应用程序并行化程度较低。尤其是: Serial Time 和 Potential Gain 分别与 Elapsed Time 和 Parallel Region Elapsed Time 相比都非常大。Serial Time 包括 SMC 的初始化时间和 5 个时间步长计算过程的串行时间。因此我们将重点关注 Potential Gain 部分, 而不是它只包含并行区域。Potential Gain 与 Parallel Region Elapsed Time 的较大比值表明了该应用程序还有可优化的空间。图中的中间部分是 CPU Usage Histogram, 图中展示了固定数量 CPU 同时运行的时间百分比。从该直方图中我们可以看出我们现在的性能与 240 倍 CPU 并发性的目标还有很大的差距。图 4-4 的最后一个部分是 CPU 的时移图, 它展示了 CPU 在时间推移过程中的使用情况。图中的第一行数据表示该应用程序的进程, 第二行数据表示进程中加载的驱动模块信息。我们发现在运行过程中 libiomp5.so 模块使用最多, 而实际工作模块 (主要包括 main.intel 等模块) 使用次数最少。这就说明了应用程序在执行过程中或者只有少量的可并行区域或者出现严重的负载不均衡。无论出现上述的哪种情况, VTune Amplifier 揭示了应用程序出现较差性能的主要问题在线程等待上, 而不是运行代码上。

通过使用 VTune Amplifier 对应用程序的性能进行热点调查分析, 我们发现了主要的热点是源文件 kernels.f90 中的 diffterm_2 子例程上 (在下面的描述中, 我们简称为 diffterm 子例程)。这个函数用于计算流体动力学 Navier-Stokes 方程的扩散项。扩散项的计算过程包含了许多循环而且这些循环中需要使用邻居单元格中的数据进行 stencil 操作。通过分析这段程序的源码, 我们发现 diffterm 中有一个较大的并行区域 (如图 4-5 所示)。虽然这个并行区域比较大, 但是它包含了具有成百上千个 OMP DO 操作的小型嵌套循环。这些细粒度的并行难以达到最佳的性能。

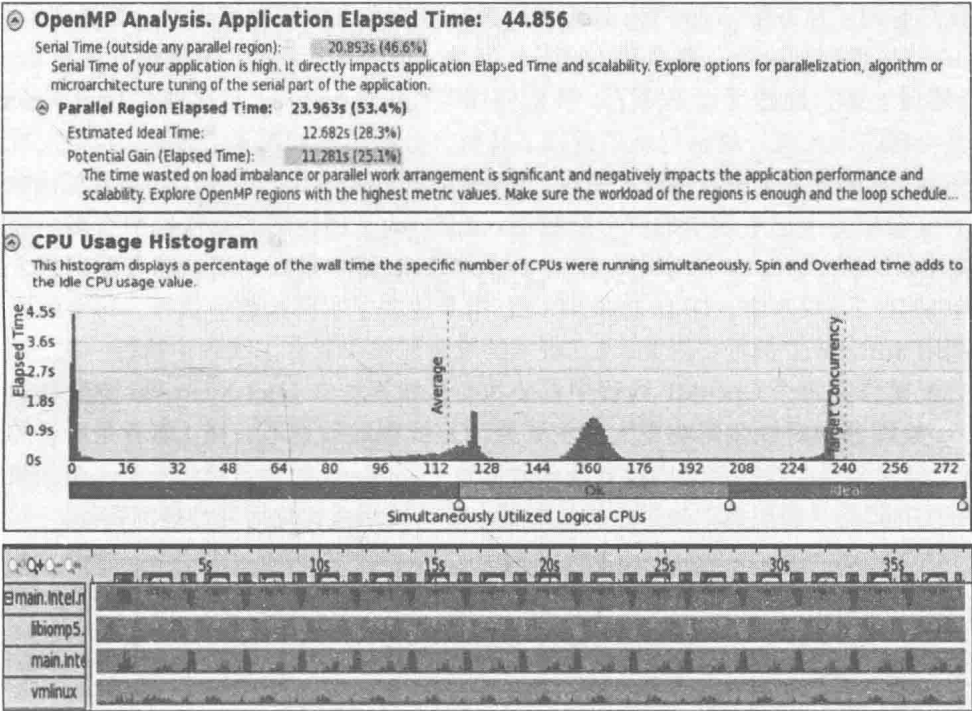


图 4-4 XMC 1.0 版本（Intel Xeon Phi 协处理器）：VTune Amphifier XE 屏幕截面，上图对应 OpenMP 分析，中图对应 CPU 使用情况直方图，下图对应一段时间内的视图

OpenMP 的工作机制导致了细粒度并行的产生。OpenMP 可并行的应用程序从主线程开始执行，当遇到并行区域时它将并行区域划分给多个工作线程，这时所有的操作都在并行执行。当 OpenMP 遇到 OMP DO 共享工作结构时，OpenMP 将循环迭代操作划分给各个线程执行。OpenMP 循环子句 COLLAPSE (n) 指定了在一个嵌套循环中，COLLAPSE (n) 次循环被折叠到一个大的迭代空间中并在各线程中进行划分。OpenMP 共享工作结构以 OMP END DO 作为结束标记，如果 NOWAIT 同步子句没有在 OMP END DO 后使用，那么在结束时会出现隐式栅栏。隐式栅栏使得所有线程完成当前的循环、结构化块或并行区域，才能继续执行后面的程序。因此，OMP DO 共享工作结构表明实际的一部分并行工作会在结束时产生隐式栅栏，这些小的并行共享工作区域将产生大量的开销和负载不均衡。一种可行的解决方法是进一步了解 OpenMP 的粗粒度并行方法。

4.3 2.0 版本：线程盒

在该版本中，我们引入了 OpenMP 的粗粒度并行方法。该方法在计算扩散项（diffterm 子例程）时几乎不需要修改源码。1.0 版本和 2.0 版本的源代码分别如图 4-5 和图 4-6 所示。图中的代码进行了精简，只保留了关注的重点部分。

在该方法中，计算区域分解为多个子区域，每个线程负责一个子区域的计算。每个线程调用 get_threadbox 函数返回当前线程的子区域的上限和下限。现在的 diffterm 子例程并不包含 OpenMP 指令，它需要定义两个新参数标识工作区域。该优化方法在 diffterm 程序中有效果，因为在不同的线程子区域中的嵌套循环互不影响。

通过扩大并行区域（并行区域不包含共享工作结构）和改善负载均衡的优化方法，我们预期应用程序在 Intel Xeon Phi 处理器和 Intel Xeon Phi 协处理器上性能将得到提升。如图 4-7

所示，我们看到了在 Intel Xeon Phi 处理器的运行性能和可扩展性得到了显著提高（1.0 版本过去是 10 倍的性能加速比，现在是 16 倍）。但是，如图 4-8 所示，该优化方法在 Intel Xeon Phi 协处理器上运行性能反而下降了。我们使用 VTune Amplifier 对这种在 Intel Xeon Phi 协处理器上出现的令人意外的运行结果进行了分析，分析结果如图 4-9 所示。我们发现了程序中的 vmlinux 模块成为了提升性能的瓶颈。vmlinux 时间随着代码中多个动态分配函数的调用（比如，tmp 数组是上述示例中的可分配数组，diffterm 子例程的实际代码中有多个这样的数组）急速增长。在 SMC 1.0 版本中，OMP PARALLEL 指令是在调用动态分配函数之后执行的。而在 SMC 2.0 版本中，OMP PARALLEL 指令放在了主调函数中执行，每个线程的子区域都要调用 diffterm 子例程。因此，1.0 版本的堆分配操作只在主线程中执行一次，而 2.0 版本的堆分配操作在每个 OpenMP 线程中都要执行。该操作在 Intel Xeon Phi 协处理器中比在 Intel Xeon 处理器中对性能影响更大的主要原因是线程的数目不一样（前者是后者的 5 倍还多）。每个线程调用的这些动态内存分配函数导致了 vmlinux 时间急速增长进一步约束了线程的可扩展性并阻碍了我们试图通过在协处理器上进行线程盒优化提升性能途径。

```

call diffterm(U,dUdt,domlo,domhi)

subroutine diffterm(U,dUdt,domlo,domhi)
  ! domlo, domhi: lower and upper bounds
  ! of the computational domain
  integer, intent(in) :: domlo(3), domhi(3)

  ! U and dUdt are four-dimensional arrays.
  ! The first three are for spatial dimensions,
  ! whereas the fourth for variable components such
  ! as density and velocity.
  ! U has four ghost cells on each side.
  real, intent(in) :: U(domlo(1)-4:domhi(1)+4,
                        domlo(2)-4:domhi(2)+4,
                        domlo(3)-4:domhi(3)+4,
                        nU)
  real, intent(inout) :: dUdt(domlo(1):domhi(1),
                              domlo(2):domhi(2),
                              domlo(3):domhi(3),
                              nU)

  ! tmp is a local array
  real, allocatable :: tmp(:, :, :)

  allocate(tmp (domlo(1)-4:domhi(1)+4,
                domlo(2)-4:domhi(2)+4,
                domlo(3)-4:domhi(3)+4) )

  !$OMP PARALLEL PRIVATE(...)

  !$OMP DO COLLAPSE(2)
  do k = domlo(3)-4, domhi(3)+4
    do j = domlo(2)-4, domhi(2)+4
      do i = domlo(1)-4, domhi(1)+4
        tmp(i,j,k) = .....
      end do
    end do
  end do
  !$OMP END DO

  ! lots of nested loops threaded with OMP DO ...

  !$OMP DO COLLAPSE(2)
  do k = domlo(3), domhi(3)
    do j = domlo(2), domhi(2)

```

图 4-5 SMC 1.0 细粒度并行代码（不是全部代码，保留重要操作的简化版代码）


```

        do i = domlo(1), domhi(1)
            dUdt(i,j,k,1) = .....
        end do
    end do
end do
!$OMP END DO

!$OMP END PARALLEL
end subroutine diffterm

```

图 4-5 (续)

```

!$OMP PARALLEL PRIVATE(lo,hi)
call get_threadbox(lo,hi)
call diffterm(lo,hi,U,dUdt,domlo,domhi)
!$OMP END PARALLEL

subroutine diffterm(lo,hi,U,dUdt,domlo,domhi)
    ! lo, hi: lower and upper bounds of the assigned
    ! sub-domain for this thread
    ! domlo, domhi: lower and upper bounds of the
    ! computational domain
    integer, intent(in) :: lo(3), hi(3)
    integer, intent(in) :: domlo(3), domhi(3)

    ! U and dUdt are four-dimensional arrays.
    ! The first three are for spatial dimensions,
    ! whereas the fourth for variable components such
    ! as density and velocity.
    ! U has four ghost cells on each side.
    real, intent(in) :: U(domlo(1)-4:domhi(1)+4,
                          domlo(2)-4:domhi(2)+4,
                          domlo(3)-4:domhi(3)+4,
                          nU)
    real, intent(inout) :: dUdt(domlo(1):domhi(1),
                                domlo(2):domhi(2),
                                domlo(3):domhi(3),
                                nU)

    ! tmp is a local array
    real, allocatable :: tmp(:, :, :)

    allocate( tmp(lo(1)-4:hi(1)+4,
                  lo(2)-4:hi(2)+4,
                  lo(3)-4:hi(3)+4) )

    do k = lo(3)-4, hi(3)+4
        do j = lo(2)-4, hi(2)+4
            do i = lo(1)-4, hi(1)+4
                tmp(i,j,k) = .....
            end do
        end do
    end do

    ! lots of nested loops without any OMP directives

    do k = lo(3), hi(3)
        do j = lo(2), hi(2)
            do i = lo(1), hi(1)
                dUdt(i,j,k,1) = .....
            end do
        end do
    end do
end subroutine diffterm

```

图 4-6 SMC 2.0 粗粒度并行代码, OpenMP 指令从 diffterm 子例程上转移到调用 diffterm 的程序上 (不是全部代码, 保留重要操作的简化版代码)

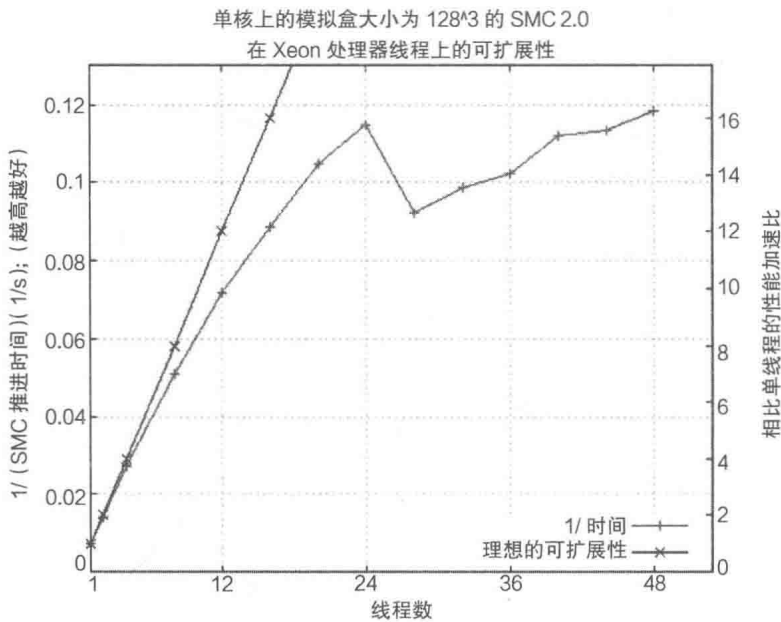


图 4-7 SMC 2.0 版本粗粒度并行代码，处理器上 OpenMP 线程执行性能和线程可扩展性得到提升

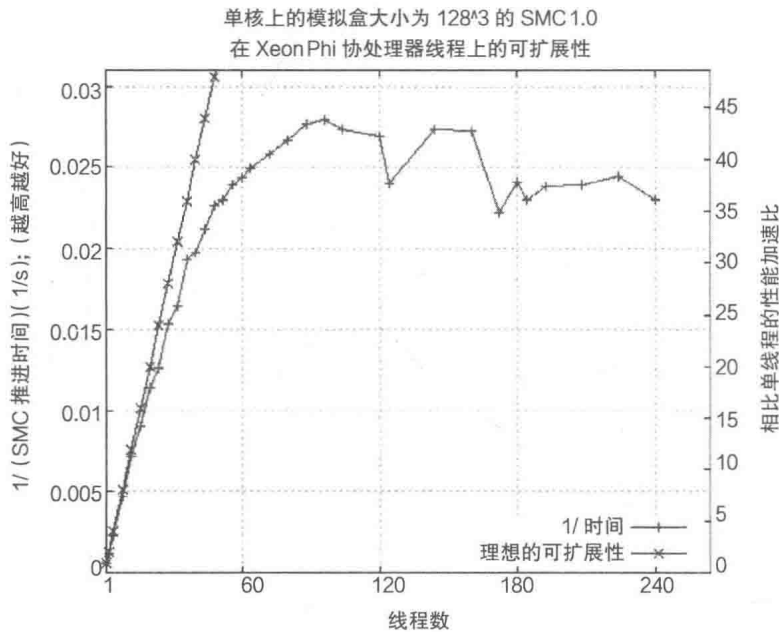


图 4-8 SMC 2.0 版本粗粒度并行代码，协处理器上 OpenMP 线程执行性能和线程可扩展性低于 1.0 版本

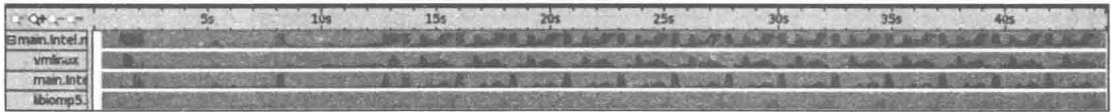


图 4-9 SMC 2.0 版本（Intel Xeon Phi 协处理器）VTune Amplifier XE 时移图截图。其中 vmlinux（第二行）与图 4-4 相比成为热点部分

4.4 3.0 版本：栈内存

在 diffterm 子例程中有许多本地数组。2.0 版本中的这些数组动态分配在堆中。这些分配操作是线程安全的但通常不可压缩。在本版本中，我们转向使用固定的数组而不是动态分配的数组存储这些局部变量。例如，上一节的示例中提到的 tmp 数组改为：

```
real ::tmp( lo(1)-4:hi(1)+4,  
            lo(2)-4:hi(2)+4,  
            lo(3)-4:hi(3)+4 )
```

这里需要注意的一点是，我们必须确保每个线程有一个足够大的栈内存空间。这可以通过“OMP_STACKSIZE”环境变量设置。

图 4-10 和图 4-11 展示了 SMC 3.0 版本的性能情况。在 Intel Xeon Phi 协处理器上，我们得到了难以置信的性能和可扩展性提升（该版本获得约 70 倍性能加速比，1.0 版本的约为 38 倍）。Xeon 处理器上的性能和可扩展性也有些许提高。但是，我们碰到了另一个问题。当应用程序在小规模线程上运行时需要设置 OMP_STACKSIZE 数组非常大，这对于一些分块类型的优化是件令人痛苦的事情。

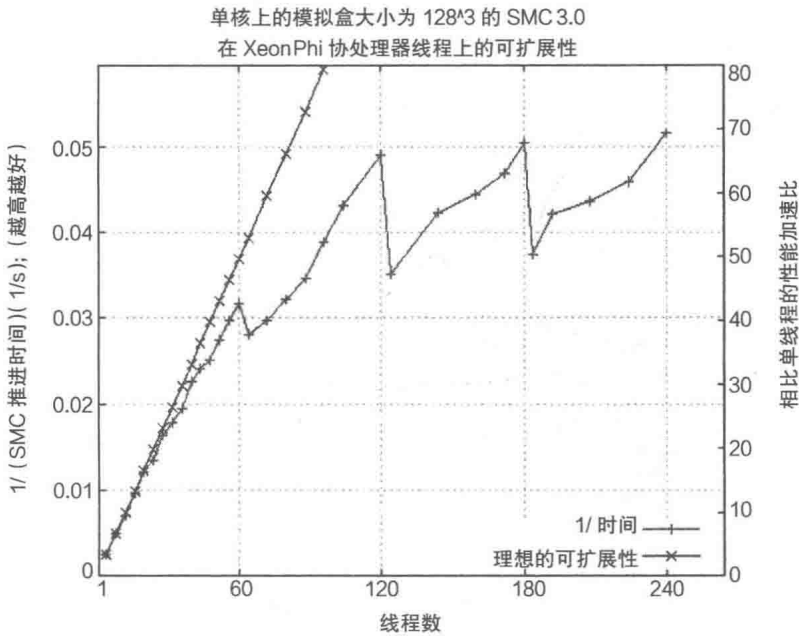


图 4-10 SMC 3.0 版本：Intel Xeon Phi 协处理器线程可扩展性，使用代替动态分配数组的固定大小数组极大地提升了 SMC 使用线程盒的性能和可扩展性

4.5 4.0 版本：分块

当使用线程的数目比较少时，3.0 版本对于每个线程需要设置一个较大的栈内存空间。为了减少这种堆栈空间大小的约束，我们将每个线程的子区域进一步划分为更小的块。块的大小可以在运行时系统参数文件 inputs_SMC 文件中设置。对分块区域进行循环操作时调用 diffterm 例程（如图 4-12 所示）。使用块的另一个优点是能够减少工作集大小，从而得到更好的缓存性能。在这里需要注意的是，这对于线程子区域在进行分块之前比目标块还

小的情况无效。

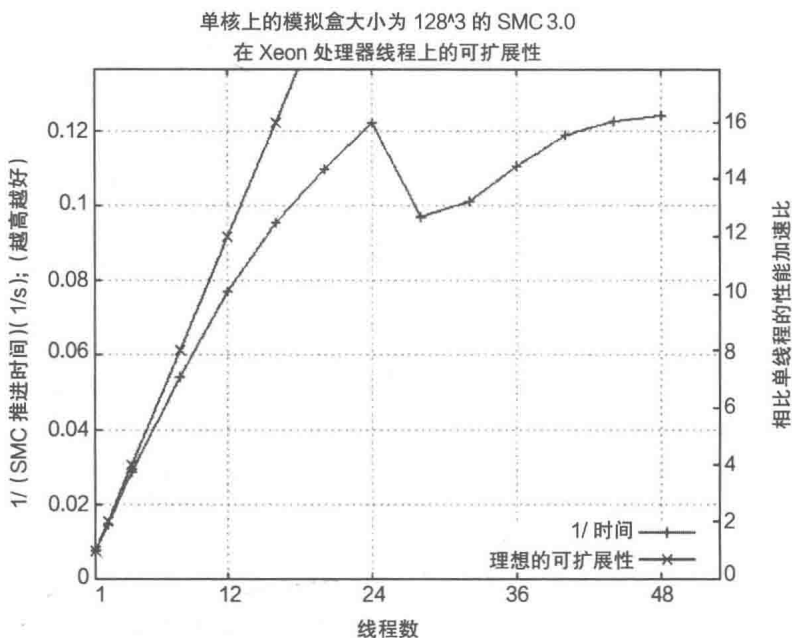


图 4-11 SMC 3.0 版本：处理器线程可扩展性与 2.0 版本在 Intel Xeon Phi 处理器上相比只有轻微的性能提升

```
Version 3.0:
!$OMP PARALLEL PRIVATE(lo,hi)
call get_threadbox(lo,hi)
call diffterm(lo,hi,U,dUdt,domlo,domhi)
!$OMP END PARALLEL

Version 4.0:
!$OMP PARALLEL PRIVATE(lo,hi,iblock,nblocks)
nblocks = tb_get_nblocks()
do iblock = 1, nblocks
  call get_threadbox(lo,hi,iblock)
  call diffterm(lo,hi,U,dUdt,domlo,domhi)
end do
!$OMP END PARALLEL
```

图 4-12 SMC 4.0 版本：将从 2.0 版本中引入的 get_threadbox 操作中的子区域划分为更小的块

图 4-13 和图 4-14 展示了分块优化后 SMC 的可扩展性。这里并没有非常大的差别。一部分实验结果变得更差，其中包括影响扩展规模（从 70 倍到 100 倍）的单线程结果，但是峰值性能仍然是一样的。虽然一些实验结果会变差，但是我们仍然保持这种变化，因为在使用小规模线程或大型化学网络计算时能够降低栈空间大小的限制。

4.6 5.0 版本：向量化

充分发挥诸如 Intel Xeon Phi 协处理器之类的众核架构性能的关键因素是使用 SIMD 指令。在当前的 SMC 简化版本中，化学反应速率计算相对简单，因为使用只有 9 种物质参与

的一个网络即可。然而，对于包含数百种物质的多个网络的计算，优化反应速率的时间开销非常重要。这是因为随着参与物质的种类数增加反应速率的计算开销比扩散项的计算开销增长更快。这里提出了一种使反应速率计算更加向量化的方法。

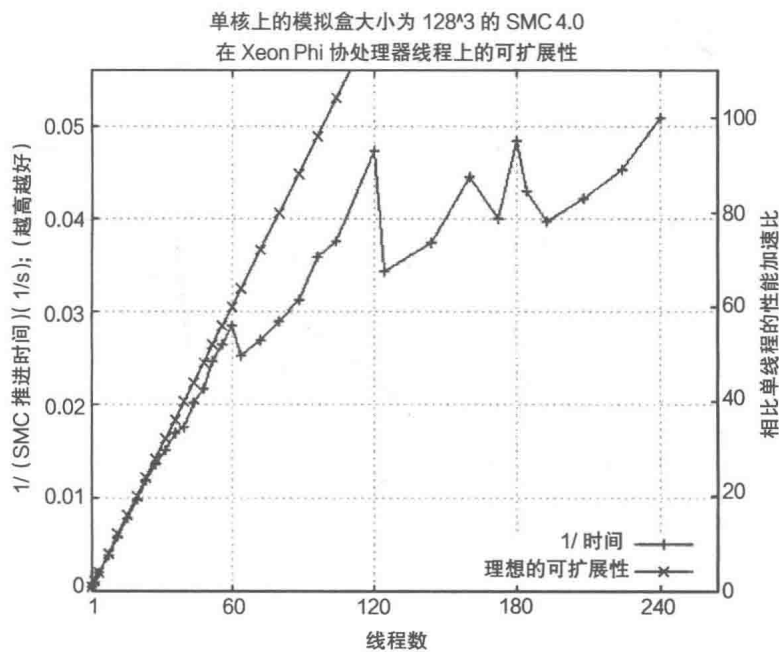


图 4-13 SMC 4.0 版本：分块后的协处理器线程可扩展性 (inputs_SMC 文件设置的参数为负 1、4、5)。图中一些点的性能有小幅度下降，但 240 个线程的性能与 3.0 版本基本一样

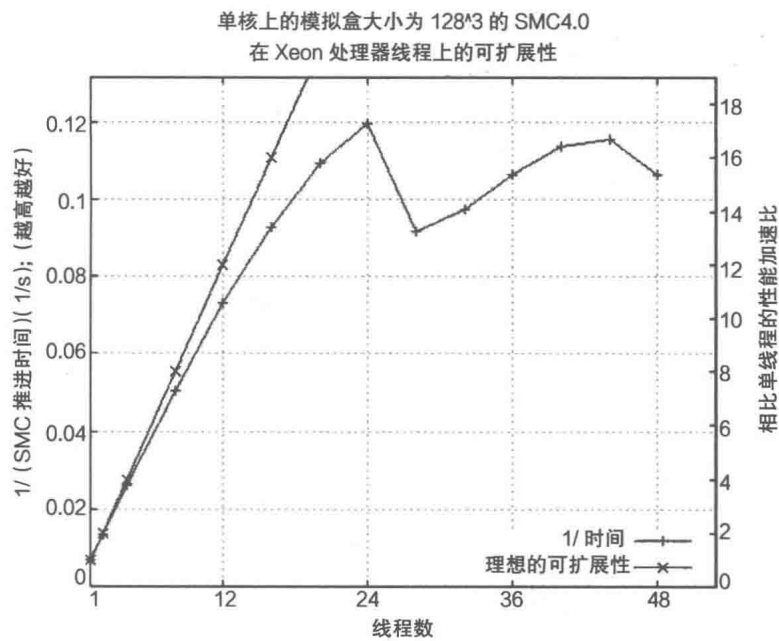


图 4-14 SMC 4.0 版本：分块后的处理器线程可扩展性 (inputs_SMC 文件设置的参数为 -1、16、16)。图中 24 个线程的实验数据与 3.0 版本的一样，但是 48 个线程的性能有所下降

原始版本的化学反应速率计算如图 4-15 所示。其中的内部循环包含了 ckwyr 子例程的调用，该子例程非常复杂且难以与其他程序内联。如图 4-16 所示，ckwyr 子例程在单个点上执行。它以密度、温度和质量分数为输入参数，返回各个物质的摩尔产率。最终的结果存储在一个四维数组中。由于 ckwyr 在一个不能内联的内部循环中执行且 ckwyr 子例程同一时刻只能处理一个单元格，因此难以实现向量化。

Original computation of Chemical Reaction Rates:

```
do k=lo(3),hi(3)
  do j=lo(2),hi(2)
    do i=lo(1),hi(1)
      ! mass fraction
      Yt = q(i,j,k,qyl:qyl+nspecies-1)
      call ckwyr(q(i,j,k,qrho),
                 q(i,j,k,qtemp), Yt, wdot)
      up(i,j,k,iryl:iryl+nspecies-1) = wdot
        * molecular_weight
    end do
  end do
end do
```

图 4-15 SMC 4.0 版本：不能向量化的原始化学反应速率计算代码

```
k_f[0] = 3.547e+15*exp(-0.406*T-
  8352.8934356925419706*invT);
k_f[1] = 50800.0*exp(2.67*T-
  3165.2328279116868543*invT);
k_f[2] = 2.16e+08*exp(1.51*T-
  1726.0331637101885462*invT);
// .....
k_f[5] = 6.165e+15*exp(-0.5*T);
// .....
k_f[9] = 1.66e+13*exp(-
  414.14731595728432012*invT);
// .....
k_f[11] = 3.25e+13;
// .....
k_f[20] = 5.8e+14*exp(-
  4809.2416750957054319*invT);
```

图 4-16 SMC 4.0 版本：原始的 ckwyr 函数代码，同一时间只能计算一个点

```
for (int i=0; i<np; i++) {
  k_f[0][i] = 3.547e+15*exp(-0.406*T[i]-
    8352.8934356925419706*invT[i]);
  k_f[1][i] = 50800.0*exp(2.67*T[i]-
    3165.2328279116868543*invT[i]);
  k_f[2][i] = 2.16e+08*exp(1.51*T[i]-
    1726.0331637101885462*invT[i]);
  // .....
  k_f[5][i] = 6.165e+15*exp(-0.5*T[i]);
  // .....
  k_f[9][i] = 1.66e+13*exp(-
    414.14731595728432012*invT[i]);
  // .....
  k_f[11][i] = 3.25e+13;
  // .....
  k_f[20][i] = 5.8e+14*exp(-
    4809.2416750957054319*invT[i]);
}
```

图 4-17 SMC 5.0 版本：修改后可向量化的化学反应速率计算代码

为了实现化学反应速率计算的向量化，我们需要修改 ckwyr 子例程，将它的密度、温度和质量分数等输入参数用向量表示。图 4-17 表示的是向量化的实现。为了使用新的实现，需要修改函数调用代码来生成输入向量 Yt 并使用 ckwyr 的输出向量 wdot，如图 4-18 所示。这种改进方法使得化学反应部分在协处理器上获得了 4 倍的性能加速比。

除了对化学反应部分进行向量化之外，Fortran 语言的 SIMD 指令用来处理 diffterm 中的多组内部循环。编译器在这里不会选择向量化，因为它们之间有潜在的依赖关系，程序的正确性由编译器来保证。

图 4-19 和图 4-20 展示了 SMC 向量化优化后的性能和可扩展性。

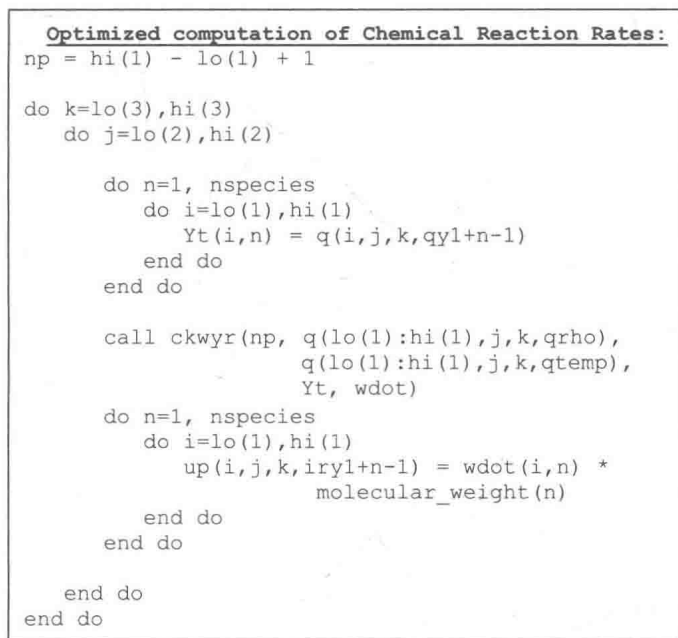


图 4-18 SMC 5.0 版本：修改后可向量化的化学反应速率计算代码

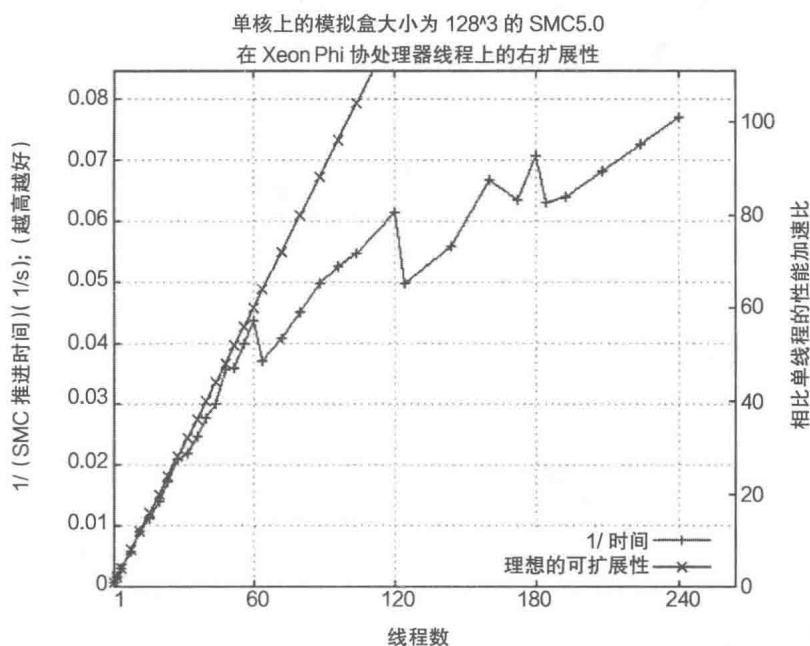


图 4-19 SMC 5.0 版本：向量化优化后协处理器线程可扩展性

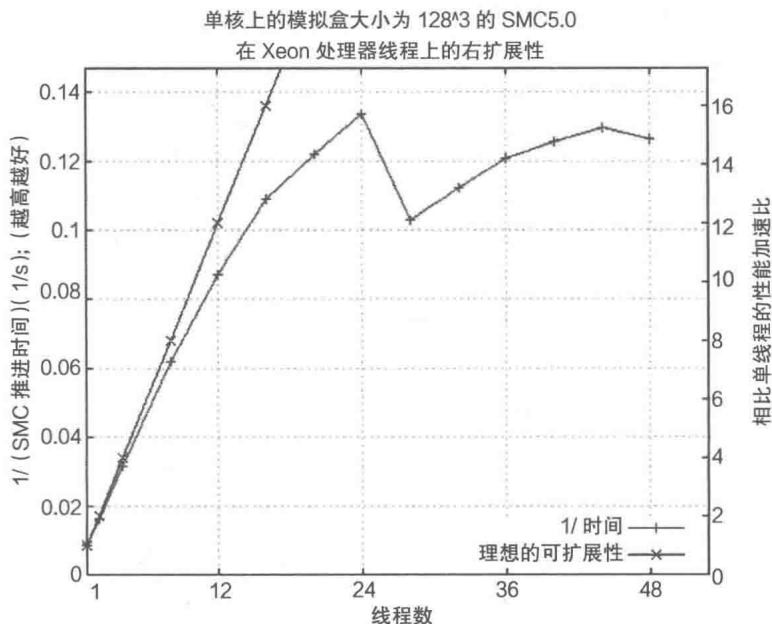


图 4-20 SMC 5.0 版本：向量化优化后协处理器线程可扩展性

注意，如图 4-19 所示，在 Intel Xeon Phi 协处理器上最好性能是 12.99s。最近我们发现了一些 Fortran 语言上的编译优化项可以让我们在协处理器上达到 9.11s（与图 4-19 相比提升了 1.43 倍），并且在处理器上保持相同的性能。提升性能的编译优化项是 `-fno-alias-no-prec-div-no-prec-sqrt-fimf-precision=low-fimf-domain-exclusion=15-align array64byte-opt-assume-safe-padding-opt-streaming-stores always-opt-streaming-cache-evict=0`。

4.7 Intel Xeon Phi 协处理器上的运行结果

我们的优化方法在 Intel Xeon Phi 协处理器和 Intel Xeon Phi 处理器均使性能得到了提升。当我们使用相同的编程技术时，同时提升了应用程序的性能和可扩展性。应用程序在 Intel Xeon Phi 协处理器上表现的可扩展性相比单线程的从 38 倍提升到 100 倍，且性能也提升了 1 倍多。在处理器上表现的可扩展性从 10 倍提升到了 16 倍，性能也提升了近 1 倍。

图 4-21 表示的是最终的 VTune Amplifier 汇总 OpenMP 性能快照。在 OpenMP 分析部分，潜在可提升的时间和串行时间都有大幅度减少。串行时间中的初始化时间无法被优化且不是 SMC 性能判断的一部分；在中间部分，我们的优化方法在 CPU 使用率直方图上实现了并发；最后，时移图中 libiomp5.so 模块时间明显降低。该工作负载的并发性得到了显著提高。

仅在性能方面，优化后的 Intel Xeon Phi 处理器性能比优化后的 Intel Xeon Phi 协处理器性能好。我们的优化工作与基本版本 1.0 相比无论在处理器上还是协处理器上都表现出更好的性能。此外，我们对当前的编程技术进行了优化，使得应用程序可扩展到更多的节点上运行。

当我们将应用程序移植到下一代 Xeon Phi（代号为 KNL）时，我们期望能获得更进一步的性能提升。Intel 对于多核和众核处理器使用的共享编程模型、编程语言和开发工具都能使我们的优化工作受益，因为我们对于当前的 KNL 的优化工作兼容了协处理器和 KNL 处理器硬件。

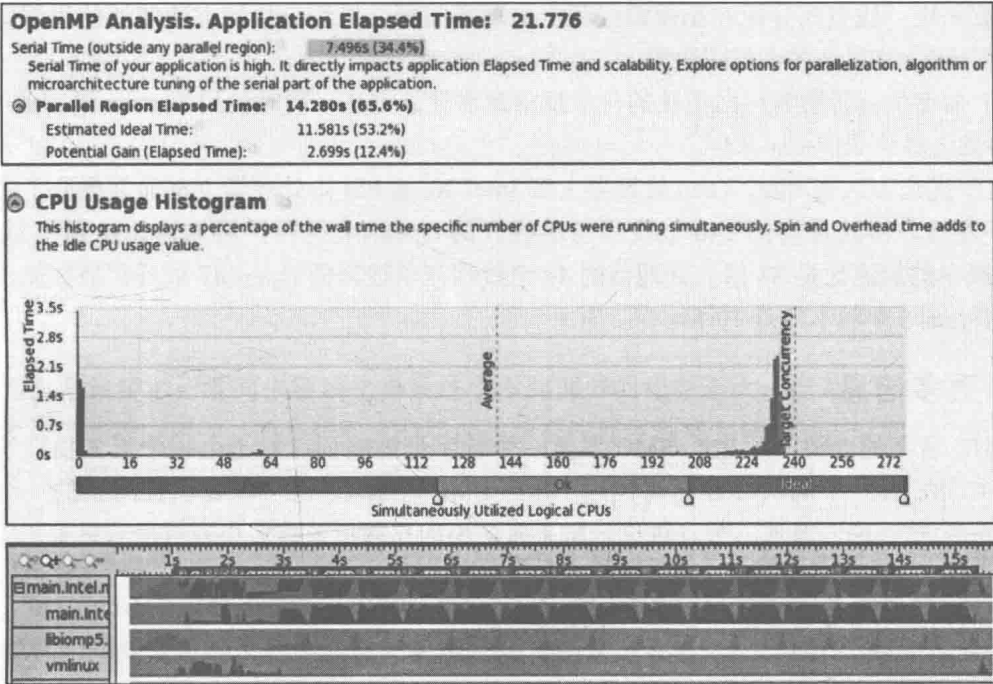


图 4-21 SMC 5.0 版本 (Intel Xeon Phi 协处理器): VTune Amplifier XE 三个不同部分的性能分析截图: OpenMP Analysis、CPU Usage Histogram 和时移图。表明了相比 SMC 1.0 版本 (图 4-4) 性能有较高的提升

4.8 总结

现如今 Intel 处理器和协处理器在每个节点上出现越来越多的内核和硬件线程。而且这种内核数和线程数不断增长的趋势在未来的处理器中不会下降。要充分利用这些并行硬件环境，集群混合架构的程序开发人员需要分析和优化并行编程的实现。这个案例研究讨论了可提高并发性的一些方法。其中包括如何分析和追踪应用程序的性能和可扩展性，帮助我们分析 OpenMP 性能及 OpenMP 的工具和向量化优化技术。

对于分析和追踪性能，我们使用了一个带有两个 y 轴坐标（一个描述的是性能，另一个描述相对于单线程的性能加速比）的 OpenMP 线程扩展性图。这一个图给我们提供了许多分析功能：版本间和平台间的性能对比，线程可扩展性改进的量化分析，以及每个线程粒度上的改进分析。

对于工具，我们使用的是带有 OpenMP 分析功能的 VTune Amplifier XE 2015 工具。VTune Amplifier 中的 OpenMP 分析功能提供了程序优化后的潜在可提升性能空间，说明了如何高效使用逻辑 CPU 以及当前在 CPU 上实际运行的程序内容。这些功能与传统的热点分析功能相结合提升了 VTune Amplifier 工具在分析和优化并行应用程序方面的能力。

我们主要做了四个方面的优化工作。

- 1) 线程盒：为了优化原始版本中的在循环层上的 OpenMP 细粒度并行性，创建一个粗粒度并行方法。
- 2) 栈空间分配：线程盒优化出现了让我们无法预料的副作用。我们通过在栈空间上分配数据的方法弥补了这个缺陷。

3) 分块: 栈空间分配在小规模线程上出现了问题。采用分块的方法能够使我们使用较小的栈空间, 尤其是在小规模线程上。

4) 向量化: 转换为可向量化的化学反应速率计算方法, 同时使用 Fortran 语言的 SIMD 指令向量化多重 diffterm 循环。

这些优化方法在 Intel Xeon 处理器上和 Intel Xeon Phi 协处理器上都显著提升了应用程序的并发性。协处理器的 240 个线程与单线程的 OpenMP 线程扩展加速比达到了 100 倍, 而 1.0 版本的加速比是 38 倍。处理器的 48 个线程与单线程的 OpenMP 线程扩展加速比达到了 16 倍, 而 1.0 版本的是 10 倍。

4.9 更多信息

SMC 名字的起源于哪里? SMC 不是一个首字母缩略词。它是由一个天文物理学家命名的。CCSE 有一个低马赫数燃烧代码 (称为 LMC)。LMC 也可以表示银河系的一个卫星星系——大麦哲伦云星系。大麦哲伦云星系有个小的兄弟星系称为小麦哲伦云星系 (SMC), 所以, SMC 燃烧代码也像是 LMC 代码的兄弟代码。

下面列举了本章推荐的一些额外的阅读材料:

- BoxLib 软件框架: <https://ccse.lbl.gov/BoxLib/>。
- Emmetta, M., Zhang, W., Bell, J.B., 2014. High-order algorithms for compressible reacting flow with complex chemistry. Combust. Theor. Model. 18(3) (<http://arxiv.org/abs/1309.7327>).
- 本章使用的 SMC 简化版本: <https://github.com/WeiqunZhang/miniSMC>。
- Intel Xeon Phi 协处理器 7120P (代号为 Knights Corner) 详情: http://ark.intel.com/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core。
- Intel Xeon CPU E5-2697 v2 (代号为 IvyBridge-EP) 详情: http://ark.intel.com/products/75283/Intel-Xeon-Processor-E5-2697-v2-30M-Cache-2_70-GHz?wapkw=e5-2697。
- Intel VTune Amplifier XE: <https://software.intel.com/en-us/intel-vtune-amplifier-xe>。
- Unat, D., Chan, C., Zhang, W., Bell, J., Shalf, J., 2013. Tiling as a durable abstraction for parallelism and data locality. In: Workshop on Domain-Specific Languages and High-Level Frameworks for HPC (<http://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp118s1.pdf>).

分阶段准同步栅栏

Jim Dempsey

美国, QuickThread Programming 公司

在电信系统中, 准同步系统是指系统中不同部分间相差不大, 但是并非完全同步。在许多大规模线程应用中, 同步线程池栅栏中的所有线程将消耗大量的处理时间。在适当的时机, 通过调整应用中的松散同步(分阶段准同步)栅栏而不是严格同步, 能够减少之前丢失的线程栅栏的一大部分等待时间。本章将这些技术应用到一个示例应用程序中。该应用程序在由 Morgan Kaufman 出版社出版并且由 Jim Jeffers 和 James Reinders 编写的 2013 版《Intel Xeon Phi Coprocessor High-Performance Programming》(HPP) 书中的第 4 章提到。

在我们描述分阶段准同步栅栏的详细使用之前, 首先对该应用程序例子的一些基础版本的优化代码进行阐述。介绍基本优化后, 讨论分阶段准同步栅栏的用法和优点, 从而融合所有技术层面使应用程序在性能方面有巨大的提升。本章旨在为读者提供一个完整的学习经验。探索的过程是最重要的, 探索到最后你将成为一个更加优秀的程序员。

需要注意的是, 本章介绍的编程优化方法同样也适用于可编程的处理器。有关主机处理器的优势将在本章的结尾部分进行讨论。

本章中系统所选用的运行时数据与 HPP 这本书中使用的略有不同。主机系统是一个采用 X79 芯片组主板且拥有一个 Intel Xeon E5-2620 处理器的工作站。此外我们还在其上安装了两个 Intel Xeon Phi 5110P 协处理器, 但是我们只使用了其中一个。HPP 书中的示例运行在 61 核协处理器上, 而本书中修改后的代码运行在 60 核协处理器上。因此, 本书中的测试和 HPP 书中使用的线程数和处理器核数略有差异。

这里不再重复 HPP 书中第 4 章的内容, 而第 4 章的结尾将作为本章的开始。正在进行的项目模拟了在立方体等 3D 容器中的溶质扩散过程。无论是原始代码还是我们修改后的代码都可以在 <http://www.lotsofcores.com> 网站上进行下载。虽然本章的内容是假定在单协处理器环境下编写的, 但是我们所使用的示例应用程序在单处理器上也可以编译、运行。

我们使用了其他章结尾的运行结果数据(如图 5-1 所示)。该图是在 60 核协处理器上使用了新测试系统的实验结果图, 是 HPP 书中图例的更新版本。图中的数据表示程序三次运行结果的平均值。目的是消除操作系统波动的影响。

注意, 图中的数字表示的是优化后版本程序的每秒百万次浮点数运算结果与单线程基版本程序的每秒百万次浮点数运算结果的比率。此外, 该图不是一个随处理器的线程数或内核数改变的可扩展图, 而是一个不同优化程序充分利用协处理器的计算能力的性能优势对比图。

图 5-1 说明了优化进程取得的进展:

- 使用正确的算法

base

单线程版本程序

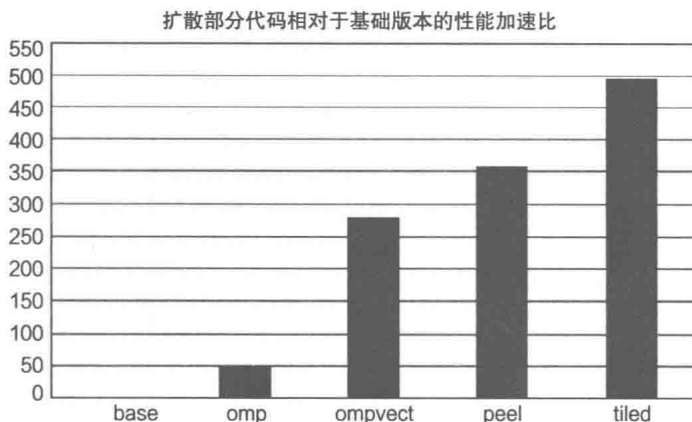


图 5-1 Xeon Phi 5110P 处理器上的相对性能提升

- 介绍并行代码

omp 并行程序简化版本

- 进一步向量化（可与添加并行代码互换）

ompvect 增加 SIMD 向量化指令

- 移除不相关代码（提高效率）

peel 从内部循环中移除不相关代码

- 提高缓存命中率

tiled 对任务进行划分以提高缓存命中率

完成前面所述的优化过程之后，如果你没有忽略特别重要的地方，那么程序已经没有多少地方需要改进了。

在前面提到的书中，作者在章节最后认为可以通过不断的努力获得额外的性能提升，即使经验丰富的并行编程程序员认为除了已调整内容外没有太大可能做进一步优化。

在列出代码改变前，强调对协处理器赋予更高的并行优化关注是非常重要的。这也就意味着不会使用一个协处理器运行一个单线程应用程序。因此，更清晰的图表对比如图 5-2 所示。图中的横坐标表示相对简单 OpenMP 并行版本的性能加速比对比情况，OpenMP 版本在图中用 omp 表示。

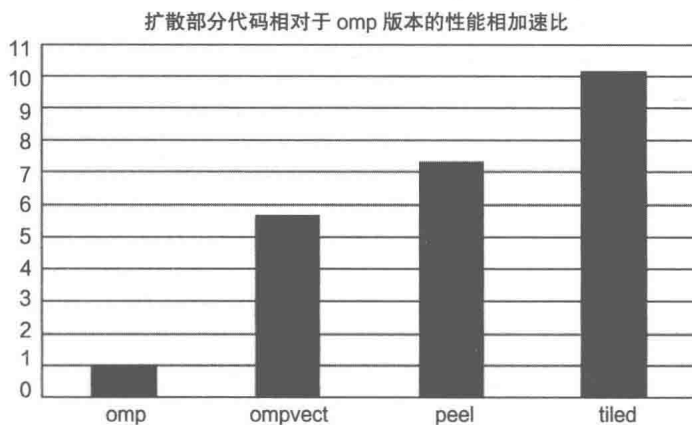


图 5-2 基于 OpenMP 的性能对比

图 5-2 明确表明了我们可以从 HPP 书中第 4 章中得到如下优化建议。下面讨论接下来

能做些什么。

首先, 做一些小范围的代码修改更容易运行上述测试。所有的改变不会影响代码的意图或性能测试情况。第一个改变是允许在 C++ 编译器命令行上增加一个 `-DNX=nnn` 的编译选项来定义 `NX` 宏。这个宏定义用来表示指定的立方体大小: `nx=NX`, `ny=NX`, `nz=NX`, 即设置问题的规模。当这个宏没有在编译器命令行选项中给出时, 将会使用默认值 256 来代替。一旦 `NX` 宏在命令行定义, 那么这个命令行值将被使用。这样做的目的是使 `Makefile` 文件不需要修改源代码即可创建一个 $256 \times 256 \times 256$ 的小数组模型和一个 $512 \times 512 \times 512$ 的大数组模型。

本书中 tiled 代码的计算部分如图 5-3 所示。

```
#pragma omp parallel
{
    REAL *f1_t = f1;
    REAL *f2_t = f2;
    int mythread;
    for (int i = 0; i < count; ++i) {
#define YBF 16
#pragma omp for collapse(2)
        for (int yy = 0; yy < ny; yy += YBF) {
            for (int z = 0; z < nz; z++) {
                int ymax = yy + YBF;
                if (ymax >= ny) ymax = ny;
                for (int y = yy; y < ymax; y++) {
                    int x;
                    int c, n, s, b, t;
                    x = 0;
                    c = x + y * NXP + z * NXP * ny;
                    n = (y == 0) ? c : c - NXP;
                    s = (y == ny-1) ? c : c + NXP;
                    b = (z == 0) ? c : c - NXP * ny;
                    t = (z == nz-1) ? c : c + NXP * ny;
                    f2_t[c] = cc * f1_t[c]
                        + cw * f1_t[c]
                        + ce * f1_t[c+1]
                        + cs * f1_t[s]
                        + cn * f1_t[n]
                        + cb * f1_t[b]
                        + ct * f1_t[t];
#pragma simd
                    for (x = 1; x < nx-1; x++) {
                        ++c; ++n; ++s; ++b; ++t;
                        f2_t[c] = cc * f1_t[c]
                            + cw * f1_t[c-1]
                            + ce * f1_t[c+1]
                            + cs * f1_t[s]
                            + cn * f1_t[n]
                            + cb * f1_t[b]
                            + ct * f1_t[t];
                    }
                }
            }
        }
    }
}
```

图 5-3 HPP 书中第 4 章的 tiled 代码


```

++c; ++n; ++s; ++b; ++t;
f2_t[c] = cc * f1_t[c]
        + cw * f1_t[c-1]
+ ce * f1_t[c]
        + cs * f1_t[s]
        + cn * f1_t[n]
        + cb * f1_t[b]
        + ct * f1_t[t];
    } // tile ny
  } // tile nz
} // block ny
REAL *t = f1_t;
f1_t = f2_t;
f2_t = t;
} // count
} // parallel

```

图 5-3 (续)

5.1 如何改善代码

计算密集性循环包含几个整数递增的单浮点数运算。使用 `-O3` 的编译器优化选项将展开循环并减少测试次数 ($x < n_x - 1$)，还可以通过使用偏移量减少 `++` 操作的次数。更重要的是，编译器会对该循环进行向量化操作。

该循环将执行 x 维 (小规模情况下是 254) 的 $n_x - 2$ 幂次。而在展开之前和之后循环将执行剩下的 x ($x=0$ 且 $x=n_x-1$) 平方次。当 $NX=256$ 时，内部循环将占到总任务的 $254/256=99.22\%$ 。

编译器在插入预取的时机和地方上面做得非常好。可以通过反汇编 (通过 `-s compiler switch`) 查看。我尝试通过使用不同的 `#pragma prefetch` 选项却得到比编译器自动预取更低性能。出于好奇，我也在本应用程序中使用 `#pragma no-prefetch` 选项，得到了非常小的性能提升 ($<1\%$)。这个可能是由于测试用例中的循环相对比较简单，处理器的硬件预取器能够为该用例提供预取。在每次循环中消除不必要的预取指令可以获得至少一个时钟周期。

HPP 书中指出最优性能通过使用 `KMP_AFFINITY=scatter` 和 `OMP_NUM_THREADS=183` 选项 (每个处理器使用 3 个线程) 获得。在 180 个线程的测试机上 (少于一个内核)，使用分块 (`YBF = 16`) 提高缓存命中率，并且我们假设书中作者通过调整这些因素获得了最佳的性能。

5.2 如何进一步改善代码

通过分析我们进一步发现了代码中存在的问题。在阅读代码以及将它与那些由多个有序的具有四个硬件线程的内核构成的协处理器计算特性相关联时，这些问题并不明显。HPP 书的作者发现每个核使用四硬件线程中的三个线程可以得到最佳性能 (对于这个应用程序)。在目标系统上运行结果显示得到了最好的数据。

第一个未在 HPP 书中使用的优化涉及每个内核中线程的 L1 和 L2 关系。下一小节将介绍如何利用这种关系。

5.3 超线程方阵

方阵这个术语是从古希腊人和古罗马军队的队列派生出来的。军队中的士兵站位是横向

肩并着肩，纵向盾挨着盾。这种方阵往前推进形成“一种不可抗拒的力量”。这里所使用的超线程方阵术语是指在一个内核中的各超线程间紧密相连并向前推进。

由于 HPP 作者在书籍出版截止日期前还要编写、测试、修改编程示例并将其插入到书稿中，因此没有在该书中讨论这种方法。但是，作者标注了更多的优化方法是可能的，所以我们将探讨这些改进方法。

第一种优化技术是将每个内核内的硬件线程压缩进一个紧耦合的工作单元（方阵）中，因此提高了 L1 和 L2 的缓存命中率。在 HPP 书的代码中，每个线程负责三维问题空间的不同部分。从 X 轴向下看（即从 Z/Y 平面），每个线程的 L1 和 L2 负责处理 X 轴每一列方向的缓存命中，如图 5-4 所示。

图 5-4 中的左侧图像表示从 X 往下看的 Z/Y 平面视角。图例说明数据有可能驻留在一些缓存中。在第一次的 X 轴方向遍历中只有中间的 c 列表现出局部的缓存命中。当计算到 $x=nx-1$ 时，c 列就有三个输入（ $c+1, c, c-1$ ）。当 $c+1$ 调入内存进行读取时，c 和 $c-1$ 也会在缓存中等待被使用。循环结束后，c,n,s,t,b 这 5 列都会出现在 L1 或 L2 缓存中（就像使用 $f1_t[c]$ 时，同一列中的 $f1_t[c+1]$ 和 $f1_t[c-1]$ 出现在缓存中一样）。

图中右侧的图像展示了下一次的 x 轴方向遍历（ $++y, x=0:n-1$ ）。需要注意的是，现在 x 轴方向 5 列中的 c 和 t 两列都处于 L1 或 L2 缓存中，而剩下的那三列不在缓存中。此外， $[c-1]$ 和 $[c+1]$ 将被调入缓存。包括每一个内核的兄弟硬件线程在内的所有线程在执行过程中都同样活跃。通过统计中间列的三次命中数据我们估算出缓存命中率为 3/7（42.86%）。这个命中率没有估算预取的情况，并且假定了 x 轴方向的长度不大于缓存的容量。

上面的内容只是一个简单的说明。由于从 Z 轴方向的向量角度来看，多个单元格将会在同一时间处理。前面的叙述只作为可视化的帮助。

5.4 关于该方案哪些地方不是最优的

首先，同一内核内的各兄弟 HT 间共享内核内的 L1 和 L2 缓存。每个内核都有自己独立的 L1 和 L2 缓存。因此原始 tiled 代码（这使用了 3/4 的可用兄弟 HT）中每个内核内的三个兄弟 HT 只能有效利用内核上 1/3 的共享 L1 和 L2 缓存。而且根据起始点的不同，兄弟 HT 可能会出现假共享并脱离相互间的热缓存队列。这不是一种高效的解决方案。

这里描述并使用了从原始 tiled 代码的最佳设置转移到不同情况下采取不同策略的方法。

注意 硬件线程（HT）和超线程（HT）的对比。从技术层面而言，Intel 将一个硬件线程称为一个 Intel Xeon 处理器的超线程。一些 Intel Xeon 处理器中对于每个内核有两个超线程，而其他的对于每个内核只有一个线程。在 Intel Xeon Phi 协处理器上，每个内核有 4 个线程，但是这是第一个使用有序处理器排序的产品（代号为 Knights Corner,KNC）。Intel 非常谨慎地宣称这些不是超线程的原因是他们想获得最佳的性能。事实上，许多 HPC 也是通过关闭超线程上工作负载的方式来获得最佳性能。这也证明了如果不是非常优秀的 HPC 用

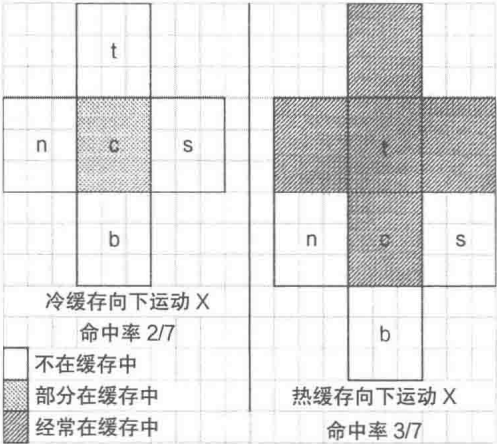


图 5-4 单线程缓存命中率分布

户，很少使用超线程来为 HPC 服务。关闭超线程是必需的，在 KNC 上的使用证明了它具有较低的性能。而且 KNC 线程对于高性能计算非常重要。其中一个重要的原因是，KNC 是一个有序的处理单元且需要额外的线程去隐藏内存延迟开销并挖掘使用向量处理器单元访问的全部潜力。下一代 Intel Xeon Phi 产品（代号为 Knights Landing, KNL）的处理单元不是有序的，所以我们有理由猜测它将会使用任何超线程技术。我们也非常纠结并意识到本章的技术在硬件线程和超线程下都可以正常使用。因此我们选择总是在说超线程并且我们最后还是会再说一次。总而言之，对于我们而言不管 HT 意味着什么，我们知道 KNC 不会使用超线程了，但是我们认为 KNL 会。我们还是会使用 HT 不管你如何理解它。

5.5 超线程方阵编码

超线程方阵是通过改变外部计算循环而引入的，使得每个内核的 HT 线程相邻的 z 个单元格在 y 轴方向和 x 轴的下方（按照向量方式）。该技术这样设计的原因会在后续内容中讨论，但是首先我们要看看选择访问模式背后的原因。对于当前特定的计算循环，这种超线程方阵设计可以提高 L1 和 L2 的缓存命中率。

我们扩展了图 5-4 的单一线程，我们将绘制出 2 路和 4 路超线程方阵的图像。这里的 2 路超线程方阵同样适用于主机处理器。

在图 5-5 的每幅图像中，左边的 c 在由内核的 HT (0) 处理的 X 列的中心位置，右侧的 c 在由内核的 HT (1) 处理的 X 列的中心位置。左边的图像表示的是 X 轴（调入页面）的冷缓存渗透， Z 移动到右侧且 Y 向下。 c 的浅色阴影表明了其中一个线程在缓存中没有命中，而相连线程访问的同一区却在缓存命中了。右侧的图像显示了 X 向下的下一步运动。注意现在估算的缓存影响。 c 和 t 的线程都命中了缓存。当每个内核分别使用三个线程和四个线程时，图像也会扩展到 $3c$ 和 $4c$ 。对于两路超线程方阵，缓存命中率的估算值为 $10/14$ (71.43%)；对于 3 路超线程方阵，缓存命中率估算值为 $16/21$ (76.19%)；而对于 4 路超线程方阵，缓存命中率估算值为 $22/18$ (78.57%)。这些图像和数据表明使用多路超线程方阵使单线程的缓存命中率明显提高了 $3/7$ (42.86%)。

超线程方阵的第二个优点是内核内的同级 HT 通信线程能够共享内核中全部的 L1 和 L2 缓存空间。它比每核单线程现有的工作空间分离技术在缓存设置上的线程工作量增加了 $2 \sim 3$ 倍。

第三个优点是它能够避免由于内核内共享出错引起的内核内线程缓存迁移问题。

实现超线程方阵的编程改变对于 OpenMP 程序员来说是非常不寻常的，但是还比较容易实现。为了使用超线程方阵删除 `#pragma omp for collapse(2)` 代码结构，并由内核与内核中的 HT 兄弟线程对迭代空间进行手动划分。这就相当于删除 5 行代码的同时插入 13 行代码。

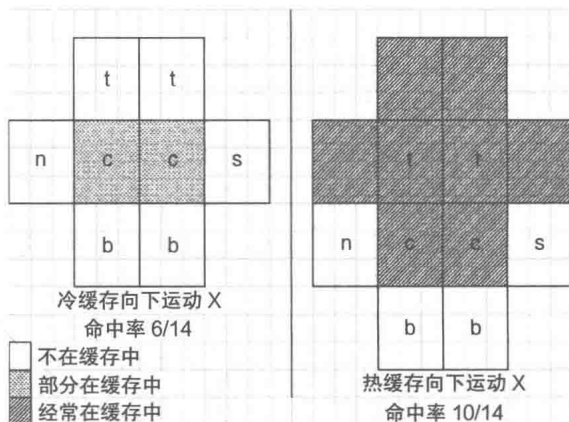


图 5-5 两路超线程方阵

5.5.1 如何确定内核间兄弟线程和内核内 HT 线程

我们可以使用程序的外部环境变量指定内核间的亲和力和放置位置，但是这可能不适用或不稳定。最好少约束和苛求环境变量。而使用一组亲和力绑定变量设置对这个功能而言可能是最好的，整个应用程序可能会从线程绑定的不同程度而受益。因此，这就需要程序确定线程绑定的亲和程度并将其应用在环境变量上。对于程序而言，更有效的方法是在处理器间协同执行线程。

我们使用 `HyperThreadPhalanx.h` 和 `HyperThreadPhalanx.c` 文件提高测试程序的性能。

上述 `HyperThreadPhalanx.c` 实用函数的主要功能有：

- 确定应用程序中最外层的 OpenMP 线程数
- 计算每个线程的逻辑内核数量（基于零且连续）
- 计算内核中每个线程的逻辑 HT 数量（基于零且连续）
- 计算逻辑内核数量
- 计算工作集中的每核 HT 数量

对于非 `HyperThreadPhalanx` 只要是合理的，应用程序可以自由选择使用相应策略。唯一合理的规则是每个内核使用相同数量的工作线程。如果不这样做，当前程序会自动选择一个最小数（通过对尚未执行的不正确配置进行测试）。

注意，程序必须在源文件中使用 `HyperThreadPhalanx`，并且 `HyperThreadPhalanx` 中的目标文件也必须和应用程序链起来。这些文件可以从 <http://lotsofcores.com> 下载。

`HyperThreadPhalanx` 目标文件以及两个线程本地存储变量 `myCore` 和 `myHT` 必须由头文件在命名空间中引入。`HyperThreadPhalanxInit()` 初始化函数在程序的开始位置只调用一次。

下面将上述函数和 HPP 书中的代码和示例结合起来。

5.5.2 超线程方阵手动分区方法

`diffusion_tiled_HT1` 的主计算函数代码如图 5-6 所示。

```
diffusion_tiled(REAL *restrict f1, REAL *restrict f2,
    int nx, int ny, int nz,
    REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
    REAL cb, REAL cc, REAL dt, int count) {
    #pragma omp parallel
    {
        REAL *f1_t = f1;
        REAL *f2_t = f2;

        // number of Squads (singles/doublets/triplets/quadruples)
        // across z dimension
        int nSquadsZ = (nz + nHTs - 1) / nHTs;

        // number of full (and partial)
        // singles/doublets/triplets/quads on z-y face
        int nSquadsZY = nSquadsZ * ny;
        int nSquadsZYPerCore = (nSquadsZY + nCores - 1) / nCores;

        // Determine this thread's triads/quads
        //(TLS init setup myCore and myHT)
        int SquadBegin = nSquadsZYPerCore * myCore;
        int SquadEnd = SquadBegin + nSquadsZYPerCore;
        if(SquadEnd > nSquadsZY)
            SquadEnd = nSquadsZY; // truncate if necessary
```

图 5-6 `diffusion_tiled_HT1` 函数


```

// benchmark timing loop
for (int i = 0; i < count; ++i) {
    // restrict current thread to its subset of Squads
    // on the Z/Y face.
    for(int iSquad=SquadBegin; iSquad<SquadEnd; ++iSquad) {
        // home z for 0'th team member for next Squad
        int z0 = (iSquad / ny) * nHTs;
        int z = z0 + myHT; // z for this team member
        int y = iSquad % ny;

        // last Squad along z may be partially filled
        // assure we are within z
        if(z < nz)
        {
            // determine the center cells
            // and cells about the center
            int x = 0;
            int c, n, s, b, t;
            c = x + y * nx + z * nx * ny;
            n = (y == 0) ? c : c - nx;
            s = (y == ny-1) ? c : c + nx;
            b = (z == 0) ? c : c - nx * ny;
            t = (z == nz-1) ? c : c + nx * ny;

            // c runs through x, n and s through y,
            // b and t through z
            // x=0 special (no f1_t[c-1])
            f2_t[c] = cc * f1_t[c] + cw * f1_t[c]
                    + ce * f1_t[c+1] + cs * f1_t[s]
                    + cn * f1_t[n] + cb * f1_t[b]
                    + ct * f1_t[t];

            // interior x's faster
            #pragma noprefetch
            #pragma simd
            for (x = 1; x < nx-1; x++) {
                ++c; ++n; ++s; ++b; ++t;

                f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1]
                        + ce * f1_t[c+1] + cs * f1_t[s]
                        + cn * f1_t[n] + cb * f1_t[b]
                        + ct * f1_t[t];

            } // for (x = 1; x < nx-1; x++)

            // final x special (f1_t[c+1])
            ++c; ++n; ++s; ++b; ++t;

            f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1]
                    + ce * f1_t[c] + cs * f1_t[s]
                    + cn * f1_t[n] + cb * f1_t[b]
                    + ct * f1_t[t];

        } // if(z < nz)
    } // for(int iSquad=SquadBegin;...)

    // barrier required because we removed implicit barrier
    // of #pragma omp for collapse(2)
    #pragma omp barrier

    REAL *t = f1_t;
    f1_t = f2_t;
    f2_t = t;
} // count
} // parallel
return;
}

```

图 5-6 (续)

实际上，我们移除了和 OpenMP 循环控制有关的 5 行代码并添加了 13 行手动控制代码（相差 8 行代码）。tilted_HT1 代码也根据阻塞因素执行情况进行了修改。为了能够支持数据流和硬件预取，我们移除了阻塞控制模块。

原始 tiled 代码和新的 tiled_HT1 代码在每个问题规模上的三次运行结果如图 5-7 所示。


```
export KMP_AFFINITY=scatter
export OMP_NUM_THREADS=180

./diffusion_tiled_xphi
118771.945
123131.672
122726.906
Average: 121543.508

./diffusion_tiled_Large_xphi
114972.258
114524.977
116626.805
Average: 115374.680

export KMP_AFFINITY=compact
unset OMP_NUM_THREADS

./diffusion_tiled_HT1_xphi
134904.891
131310.906
133888.688
Average: 133368.162

./diffusion_tiled_HT1_Large_xphi
118476.734
118078.930
118157.188
Average: 118237.617
```

图 5-7 diffusion_tiled_HT1 的运行数据

实验结果表明，超线程方阵策略确实在小型模型上有了一些提高，但是在大型模型没有改进。此外，小型模型上的优化效果也不像预期那样明显。tiled_HT1 代码如图 5-8 所示。

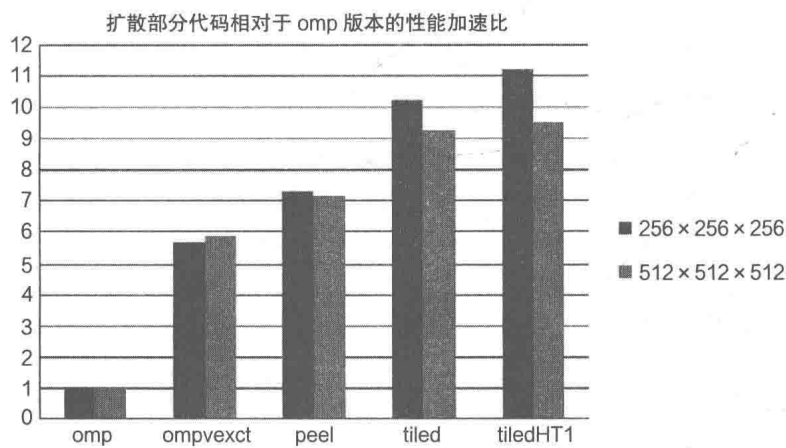


图 5-8 代码对齐对于性能的影响

虽然小型模型上的提升看起来还不错，但是这不适用于大型模型。我们将分析一下具体原因。

5.5.3 吸取教训

早期的研究是从这里开始优化的。然而，重新查阅这段代码我们又发现了新的东西。在这一点上，出现的分歧帮助我们从经验中不断成长。

测试系统是安装在不同硬盘上的 Windows 7 和 CentOS Linux 的双系统。目的是确定在任何操作系统上不会对运行原生的协处理器造成影响，并且期望应该没有明显区别。

配置环境变量用于分别在 CentOS 和 Windows 7 上运行 3 级方阵（图 5-8 针对的是 4 级方阵）。出乎意料的是，在 Windows 系统上运行相同的代码并与在 Linux 系统协处理器上的运行结果相比，相对性能数据是相反的！在更快的 Linux 主机上会快些，在更慢的 Windows 上会慢些，反之亦然。得到了毫无意义的结果。

我们突然有个想法，是不是数组分配的内存对齐问题导致的？这是根据对于 Intel64 平台和 IA32 平台多年的研究经验得出的。因此，我们用一个 printf() 函数输出缓冲区地址。结果显示 tiled 和 tiled_HT 这两种方案都是 16 字节对齐，并在页面中出现大致相同的偏移量，所以数据对齐的差异不是主要原因。但是非常奇怪的是，输出的两个缓冲区地址后，性能数据依然是相反的。运行结果如图 5-9 所示。

```
[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
f1 = 0x7fbbf3945010 f2 = 0x7fbbef944010 printf
FLOPS      : 122157.898 (MFlops) With printf
[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
f1 = 0x7ffaf7c0b010 f2 = 0x7ffaf3c0a010 printf
FLOPS      : 123543.602 (MFlops) With printf
[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
f1 = 0x7f3afa480010 f2 = 0x7f3af647f010 printf
FLOPS      : 123908.375 (MFlops) With printf
Average with printf: 123203.292 MFlops

[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
FLOPS      : 114380.531 (MFlops) Without printf
[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
FLOPS      : 121105.062 (MFlops) Without printf
[Jim@Thor-mic0 tmp]$ ./diffusion_tiled_xphi
FLOPS      : 116298.797 (MFlops) Without printf
Average without printf 117261.463 Mflops

With printf +5941.829 MFlops (+5.1% over without printf)
```

图 5-9 相对于 Open MP 的 diffusion tiled_HT1 的性能加速比

显然，由于代码位置偏移出现了 5% 的性能偏移。

这就意味着原始 tiled 代码与新的 tiled_HT 代码（之前版本）间相对性能差异性完全被代码位置的偶然性缺失所掩盖。一个版本的性能可能增加 5%，另一个也可能减少 5%，这种性能的差异高达 10%。这种差异是对于特定的示例代码而言的。不是所有的代码出现这样的性能差异都是由代码位置引起的。

但是，下次使用这样的例子时一定要考虑在调整过程中出现的代码对齐问题。

5.6 回到工作

改进后的性能并未达到预期的效果。我们在小型模型上得到了 9.7% 的性能提升，而在大型模型上仅得到了 2.5% 的性能提升。

运用奥卡姆剃刀定律：如果我们没有发现 L1 的命中率增加，那么它就没有发生过。

L1 缓存命中率可预料的提升是建立在特定算法中 L1 缓存的计算数据量大小的基础上的。

3 路小规模数据：

$$8 \times 256 \times 4 = 8 \text{ KB 低位}, 11 \times 256 \times 4 = 11 \text{ KB 高位}$$

4 路小规模数据：

$$10 \times 256 \times 4 = 10 \text{ KB 低位}, 14 \times 256 \times 4 = 14 \text{ KB 高位}$$

两个计算结果表明在 L1 的缓存中有足够的空间。一些其他优化内容可以继续做并需要进一步的深入研究（本章不讨论）。

在不考虑上述讨论内容的情况下，每秒 133 亿次的浮点运算能力远远低于 Intel Xeon Phi 协处理器的计算能力。

现在我必须思考这个问题：这个优化策略中性能下降的地方都有哪些？有什么方法可以改进？有两件事情值得关注，一个是显而易见的，另一个是不明显的。

5.7 数据对齐

很明显，向量化程度随着对齐的数据同步提高。大多数编译器会检查程序中的循环代码，如果有利于性能提升，编译器将插入前导码，前导码主要用于数据对齐的测试和对齐后的执行。这个过程称为剥离过程。在此过程中，编译器插入那些使用更高效的数据对齐方式执行的代码。最后，在剩下的未对齐数据中插入后置码以完成数据处理过程。

这听起来相当简单直到你看到内部循环（见图 5-10）。

在图 5-10 的代码中，`f1_t[c-1]` 和 `f1_t[c+1]` 两部分将会陷入向量对齐测试中。因为 `[c-1]`、`[c]` 和 `[c+1]` 永远不会同时进行数据对齐。

编译器的作者是否足够聪明来为这样的循环过程提供一些具体的优化措施呢？事实证明，他们可以。

```
#pragma simd
for (x = 1; x < nx-1; x++) {
    ++c; ++n; ++s; ++b; ++t;
    f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1]
              + ce * f1_t[c+1] + cs * f1_t[s]
              + cn * f1_t[n] + cb * f1_t[b]
              + ct * f1_t[t];
}
```

图 5-10 diffusion_...文件的内层循环

由于初始化过程充满未知，代码必须在前导码和后置码以及在减少内部循环代码的迭代数目上做更多的工作。

要特别注意的地方是，输入数组 `f1_t` 有 7 种不同的索引方法。这就意味着确定对齐信息的前导码可能要在最短时间内对 7 种索引进行排序，而索引数字最大的那个已经被向量对齐了。这对于编译器代码生成和为额外开销开辟潜在区域是比较容易完成的。

5.7.1 尽可能使用对齐的数据

在 `tiled_HT2` 程序中对数据对齐的寻址方式做了额外的优化工作。

首先，使用 `REAL` 的倍数作为 `NX` 维填充了缓冲行。这个要求是合理的。原始示例代码中使用了值 256。`NX` 对于浮点数必须为 16 的倍数，对于双精度数必须为 8 的倍数，这个要求是合理的。

为了保证数据对齐，在 `malloc` 函数的基础上，使用了带参数的 `using_mm_malloc` 函数按照缓存行大小（64）对数组进行了分配。这是一个相对简单的改变。（在接下来同样影响分配的优化后将会继续介绍这一点。）

接着，现在 `NX` 总是缓存行的偶数倍了，且数组已经与缓存行对齐了，我们构造了一个函数处理最内层循环。在该循环中，我们已经知道了数组引用中有 6 个已与缓存行对齐且两个未对齐（额外的引用指输出数组）。这两个未对齐的引用指向了 `[c-1]` 和 `[c+1]`。这样，编译器可以提前知道哪些引用是对齐的和哪些是未对齐的，不需要通过插入代码来确定。也就是说，编译器优化能够减少甚至完全消除前导码和后置码引起的时间开销。

5.7.2 冗余未必是件坏事

第二个（非显而易见的）改进是通过冗余地处理 `x=0` 和 `x=nx-1` 获得额外的优化，就

像这些单元格位于正在处理的并行吸管内一样。这就意味着可以跳过未对齐的循环的前导码和后置码部分，并且 $x=1:15$ 元素可以直接作为对齐的向量处理（与逐步计算或未对齐的向量计算相反）。我们可以对这 16 个元素做相同的处理，但是最后一个元素（ $x=nx-1$ ）的计算与其他向量元素有些不同。也就是说，在对 $x=0$ 和 $x=nx-1$ 完成错误值计算（为了释放内存）之后，我们需要执行标量计算以将正确值插入 X 列中。事实上，我们将 16 次迭代的两次标量循环转换为在 L1 缓存上的两次操作（1 次 16 路向量操作 + 1 次标量操作）。

添加必要的冗余主要包括给数组的两个向量元素分配比实际需求更大的数组空间，并返回第二个向量的指针地址。此外，需要将前一个元素值和负载数组大小后面的一个元素值清零以便执行“第一次接触”写操作。分配内存操作给可寻址内存（不能作为空数据使用）的两次额外向量提供内存。由页面位置和分配程度决定是否进行“第一次接触”写操作，但如果不做“第一次接触”，写可能会导致页面出错。

修改分配后的代码执行结果如图 5-11 所示。

```
// align the allocations to cache line
// increase allocation size by 2 cache lines
REAL *f1_padded = (REAL *)_mm_malloc(
    sizeof(REAL)*(nx*ny*nz + N_REALS_PER_CACHE_LINE*2),
    CACHE_LINE_SIZE);

// assure allocation succeeded
assert(f1_padded != NULL);

// advance one cache line into buffer
REAL *f1 = f1_padded + N_REALS_PER_CACHE_LINE;
f1[-1] = 0.0; // assure cell prior to array not NaN
f1[nx*ny*nz] = 0.0; // assure cell following array not NaN
// align the allocations to cache line
// increase allocation size by 2 cache lines
REAL *f2_padded = (REAL *)_mm_malloc(
    sizeof(REAL)*(nx*ny*nz + N_REALS_PER_CACHE_LINE*2),
    CACHE_LINE_SIZE);

// assure allocation succeeded
assert(f2_padded != NULL);

// advance one cache line into buffer
REAL *f2 = f2_padded + N_REALS_PER_CACHE_LINE;
f2[-1] = 0.0; // assure cell prior to array not NaN
f2[nx*ny*nz] = 0.0; // assure cell following array not NaN
```

图 5-11 对齐和填充分配

作为额外的优化，编译器能够使用混合乘法和增加指令生成更多代码。

tilted_HT2 代码如图 5-12 所示。

```
void diffusion_tiled_aligned(
    REAL*restrict f2_t_c, // aligned
    REAL*restrict f1_t_c, // aligned
    REAL*restrict f1_t_w, // not aligned
    REAL*restrict f1_t_e, // not aligned
    REAL*restrict f1_t_s, // aligned
    REAL*restrict f1_t_n, // aligned
    REAL*restrict f1_t_b, // aligned
    REAL*restrict f1_t_t, // aligned
    REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
    REAL cb, REAL cc, int countX, int countY) {
    __assume_aligned(f2_t_c, CACHE_LINE_SIZE);
    __assume_aligned(f1_t_c, CACHE_LINE_SIZE);
    __assume_aligned(f1_t_s, CACHE_LINE_SIZE);
```

图 5-12 diffusion_tiled_HT2.c 代码摘录


```

__assume_aligned(f1_t_n, CACHE_LINE_SIZE);
__assume_aligned(f1_t_b, CACHE_LINE_SIZE);
__assume_aligned(f1_t_t, CACHE_LINE_SIZE);

// countY is number of squads along Y axis
for(int iY = 0; iY < countY; ++iY) {
    // perform the x=0:N_REALS_PER_CACHE_LINE-1
    // as one cache line operation.
    // On Phi, 16-wide vector with one iteration
    // On AVX 8-wide vector with two iterations
    // On SSE 4-wide vector with four iterations
    #pragma noprerefetch
    #pragma simd
    for (int i = 0; i < N_REALS_PER_CACHE_LINE; i++) {
        f2_t_c[i] = cc * f1_t_c[i] + cw * f1_t_w[i]
            + ce * f1_t_e[i] + cs * f1_t_s[i]
            + cn * f1_t_n[i] + cb * f1_t_b[i]
            + ct * f1_t_t[i];
    } // for (int i = 0; i < N_REALS_PER_CACHE_LINE; i++)
    // now overstrike x=0 with correct value
    // x=0 special (no f1_t[c-1])
    f2_t_c[0] = cc * f1_t_c[0] + cw * f1_t_w[1]
        + ce * f1_t_e[0] + cs * f1_t_s[0]
        + cn * f1_t_n[0] + cb * f1_t_b[0]
        + ct * f1_t_t[0];
    // Note, while we could overstrike x=[0] and [nx-1]
    // after processing the entire depth of nx doing so
    // will result in the x=0th cell being evicted from L1.
    // Do remainder of countX run including incorrect value
    // for i=nx-1 (countX-1)
    #pragma vector nontemporal
    #pragma noprerefetch
    #pragma simd
    for (int i = N_REALS_PER_CACHE_LINE; i < countX; i++) {
        f2_t_c[i] = cc * f1_t_c[i] + cw * f1_t_w[i]
            + ce * f1_t_e[i] + cs * f1_t_s[i]
            + cn * f1_t_n[i] + cb * f1_t_b[i]
            + ct * f1_t_t[i];
    } // for (int i = 0; i < N_REALS_PER_CACHE_LINE; i++)
    // now overstrike x=nx-1 with correct value
    // x=nx-1 special (no f1_t[c+1])
    int i = countX-1;
    f2_t_c[i] = cc * f1_t_c[i] + cw * f1_t_w[i-1]
        + ce * f1_t_e[i] + cs * f1_t_s[i]
        + cn * f1_t_n[i] + cb * f1_t_b[i]
        + ct * f1_t_t[i];
    // advance one step along Y
    f2_t_c += countX; f1_t_c += countX; f1_t_w += countX;
    f1_t_e += countX; f1_t_s += countX; f1_t_n += countX;
    f1_t_b += countX; f1_t_t += countX;
} // for(int iY = 0; iY < countY; ++iY)
} // void diffusion_tiled_aligned(

diffusion_tiled(REAL *restrict f1, REAL *restrict f2,
int nx, int ny, int nz,
REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
REAL cb, REAL cc, REAL dt, int count) {
    #pragma omp parallel
    {
        REAL *f1_t = f1;
        REAL *f2_t = f2;
        // place squads across z dimension
        int nSquadsZ = (nz + nHTs - 1) / nHTs;

        // number of full/partial squads on z-y face
        int nSquadsZY = nSquadsZ * ny;
        int nSquadsZYPerCore = (nSquadsZY + nCores - 1) / nCores;

        // Determine this thread's squads
        int SquadBegin = nSquadsZYPerCore * myCore;
        int SquadEnd = SquadBegin + nSquadsZYPerCore;
        if(SquadEnd > nSquadsZY)
            SquadEnd = nSquadsZY;

        for (int i = 0; i < count; ++i) {

```

图 5-12 (续)


```

int nSquads;
// restrict current thread to its subset of squads
// on the Z/Y face.
for(int iSquad = SquadBegin; iSquad < SquadEnd;
    iSquad += nSquads) {
    // determine nSquads for this pass
    if(iSquad % ny == 0)
        nSquads = 1; // at y==0 boundary
    else
        if(iSquad % ny == ny - 1)
            nSquads = 1; // at y==ny-1 boundary
        else
            if(iSquad / ny == (SquadEnd - 1) / ny)
                nSquads = SquadEnd - iSquad; // within 1:ny-1
            else
                nSquads = ny-(iSquad%ny)-1; // iSquad%ny:ny-1

    int z0 = (iSquad / ny) * nHTs; // home z for HT(0)
    int z = z0 + myHT; // z for HT(myHT)
    int y = iSquad % ny;
    // last squad along z may be partially filled
    // assure we are within z
    if(z < nz) {
        int x = 0;
        int c, n, s, b, t;
        c = x + y * nx + z * nx * ny;
        n = (y == 0) ? c : c - nx;
        s = (y == ny-1) ? c : c + nx;
        b = (z == 0) ? c : c - nx * ny;
        t = (z == nz-1) ? c : c + nx * ny;
        diffusion_tiled_aligned(
            &f2_t[c], &f1_t[c], // aligned
            &f1_t[c-1], &f1_t[c+1], // unaligned
            &f1_t[s], &f1_t[n], &f1_t[b], &f1_t[t], // aligned
            ce, cw, cn, cs, ct, cb, cc, nx, nSquads);
    } // if(z < nz)
} // for(int iSquad = SquadBegin; ...
// barrier required because we removed implicit
// barrier of #pragma omp for collapse(2)
#pragma omp barrier
// swap buffer pointers
REAL *t = f1_t;
f1_t = f2_t;
f2_t = t;
} // count
} // parallel
return;
} // diffusion_tiled

```

图 5-12 (续)

图 5-13 中的性能图体现了两项新的程序——diffusion_tiled_HT1 和 diffusion_tiled_HT2。

从图 5-13 中能够清楚地看到，diffusion_tiled_HT2 已经开始发挥作用，至少在小规模模型上又获得了 9.5% 的性能提升。但是需要注意的是，代码调整仍然存在问题，且图中并未考虑这一点。

那么还有哪些可以优化的地方呢？

5.8 深入讨论分阶段准同步栅栏

现在我们将深入讨论本章的主题：分阶段准同步栅栏。

对于这种优化方法，我们需要使用一个合适的描述性词语解释“准同步”和“松散同步”线程的行为。

在电信系统中，准同步系统是指系统中各部分相差不大，但是并非完全同步。

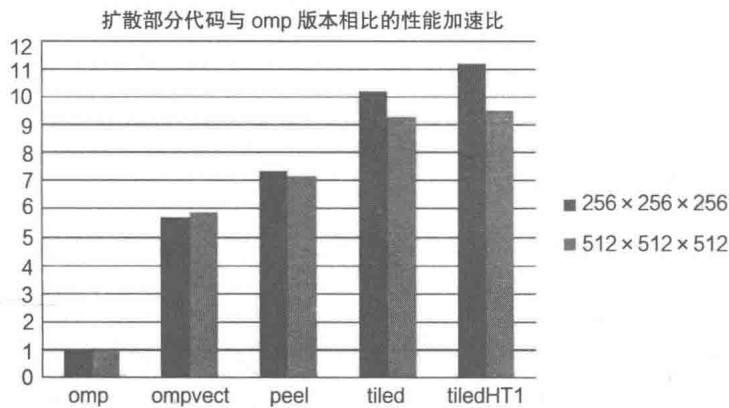


图 5-13 diffusion_tiled_HT2

在任何大规模多线程应用程序中，同步线程池栅栏与所有线程浪费了大量的处理资源。在模拟扩散过程和其他应用程序中，模拟过程需要经过多个时间间隔才能显示出结果。示例的应用程序中表示的是整个模拟的运行时间。在典型模拟应用程序中，提前设置模拟器的时间间隔为 N ，再记录或显示当前状态。然后再模拟另一个大小为 N 的时间间隔。

diffusion_tiled_HT2 代码中使用了小规模模型 ($256 \times 256 \times 256$)，4 个线程组成一个方阵。64 \times 64 的方阵沿着 Y 方向以 256 步长等分整个 Z 方向。这就表示沿着 X 方向分布着 16384 个点。每个内核 (60) 在 X 方向上平均划分得到 $16394/64$ (273.0666...) 个点。因为点的个数不是整数，所以不能保证所有内核分配到相同数量的点。因此我们在 56 个内核上平均分配了 273 个点，剩下的 4 个内核平均分配了 274 个点。各内核之间的平均工作负载差别只有 $1/273$ 。这就是我们通常所说的成本开销。

如果只有这些成本开销问题就简单多了。实际上，在 60 个内核的 240 个线程上，我们难以保证所有线程从一个并行区域同时开始，更不能保证所有线程在该区域中同时结束。(即使它们在名义上执行了相同的负载)。正如本章前面提到的，通过增加指令的方式获得所有线程在 do-work 部分和栅栏部分的使用时间，测试结果表明对于小规模问题 ($256 \times 256 \times 256$) 大约有 25% 的计算时间用于栅栏。该部分的时间开销太高了。

那么，如何减少栅栏时间呢？

我们构造了一个由每核 4 线程的双核处理器与 $16 \times 16 \times 16$ 大小数组 (忽略向量) 组成的简化结构图。当第一个线程开始执行同步操作时，tiled_HT2 代码实现了并行区域开始阶段的零误差与相同的计算时间 (栅栏处的时间开销为 0)，如图 5-14 所示。

图中左侧区域是编号为 0 的处理器内核负责的区域 (HT 线程 0 ~ 3 跨越了每个条带的宽度)，右侧区域为编号为 1 的处理器内核负责的部分 (HT 线程 0 ~ 3 跨越了每个条带的宽度)。理想的情况是两个内核同时开始并且同时结束。

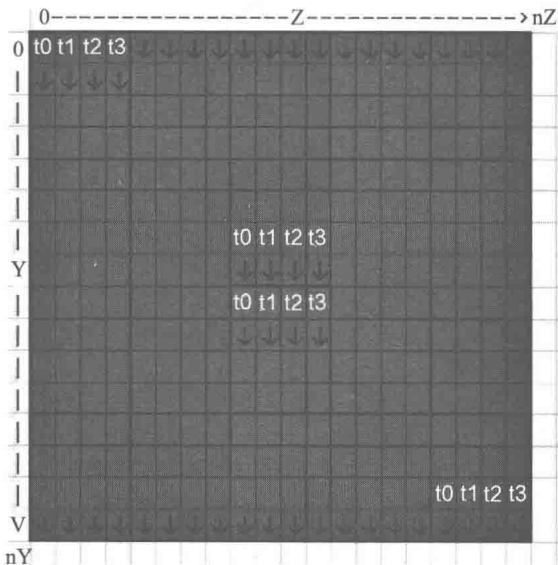


图 5-14 同步结束

真实的模拟情况如图 5-15 所示。

其中白色单元格是当第一个线程的负载运行结束后尚未处理的单元格部分。

在图 5-16 中，第一个线程到达栅栏位置（图中右下角的 t3）并等待剩余进程运行结束。然后第二个线程到达栅栏位置等待其他进程，以此类推，直到所有进程运行结束，开始执行同步。在这里使用越多的线程就意味着栅栏中的等待时间越长。

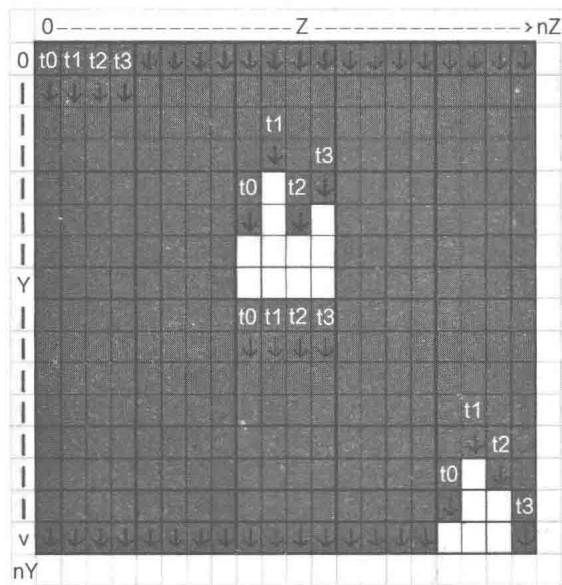


图 5-15 解决线程结束循环时的负载不均衡问题

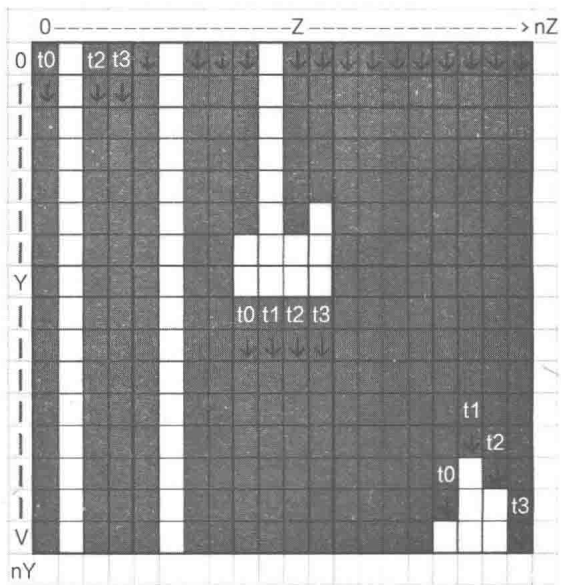


图 5-16 抢占进程负载不均衡

极端情况下的例子如图 5-16 所示。

在图 5-16 中，当第二个内核的第一个 t3 线程到达栅栏位置时，第一个内核的 t1 线程还没有开始执行它的工作。这种情况是由于存在其他进程运行在该硬件线程上造成的。也有可能是操作系统正在运行整理工作而引起的。

无论是在正常的情况下还是异常的情况下，都会有大量的时间消耗在同步上。

5.9 如何节省时间

我们使用的解决方案是将处理器内核划分成包含 1 个 Z 方向（本例中有 4 个全方阵阵组）、ny 个 Y 方向和 nx 个 X 方向的片状结构。

片状结构的左侧和右侧区域代表了每个内核第一次传输时（设置为双核，每核 4 个线程）分配的工作区域。

第一个线程负载完成的工作状态如图 5-17 所示。

现在，由单个内核栅栏代替了单个线程池的栅栏。当内核的某个线程第一次执行完

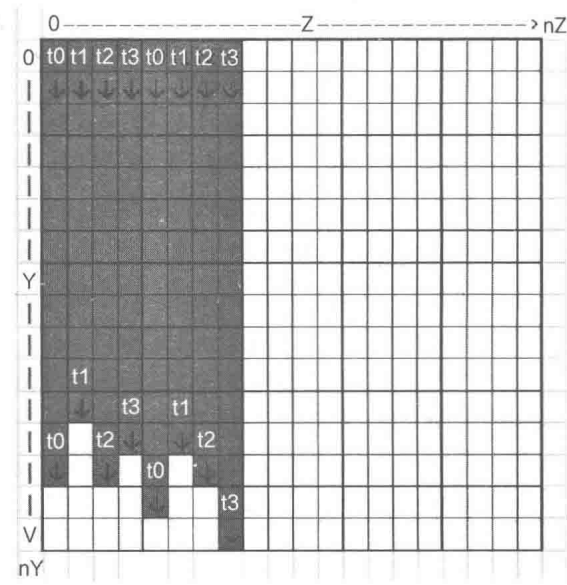


图 5-17 方阵在 Y 方向的运行情况

成后，该内核中其他线程的完成状态如图 5-18 所示。

此时，这种方式没有采用等待线程池中所有线程同时到达栅栏并进行同步的操作（在这个简化图中是指左侧的 4 个线程），完成进程栅栏的第一个线程可以继续处理下一个可用的 Z 组，并且能够在其他进程到达栅栏之前开始处理。当 0 号进程完成内核栅栏时，它的状态如图 5-19 所示。

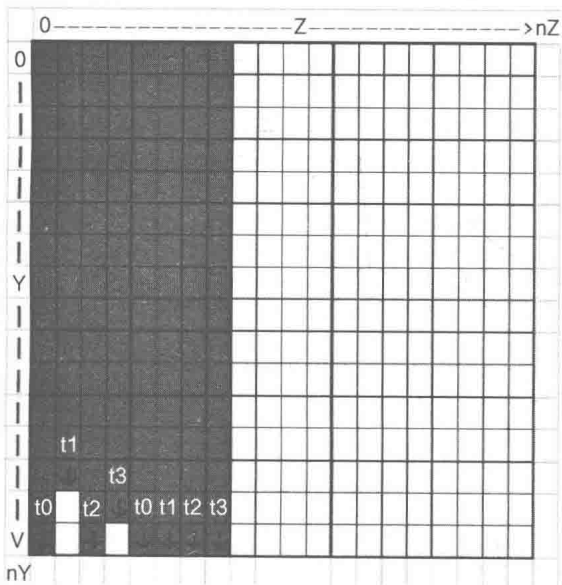


图 5-18 第一个内核在 Y 方向的运行完成情况

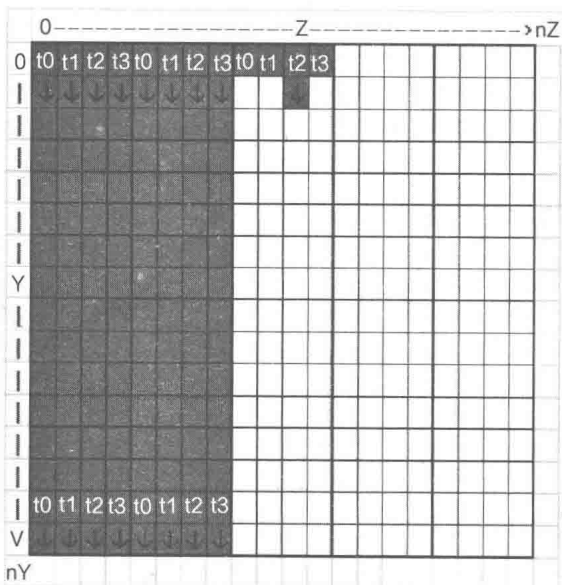


图 5-19 第二个内核完成工作

执行过程：当一个进程处理完成后，挑选下一个 Z 组，以模计数器的形式继续执行（从 0 到 $nz-1$ ）。该模计数器包含了溢出次数。

使用准同步栅栏的最大优点是可以时刻标记代码中传统线程池栅栏出现的位置和状态。它位于整个数组循环（通常称为框架）的结束部分。

在原始的 OpenMP 程序中，我们标记隐式栅栏的 `#pragma omp parallel` 的结尾位置，而在修订的 `diffusion_tiled_HT1` 和 `diffusion_tiled_HT2` 程序中同样标记了隐式 `#pragma omp barrier` 位置。

准同步栅栏技术能够使得内核结束一个框架后进入下一个框架，只要满足下一个框架的依赖关系。详细解释如图 5-20 所示。

在图 5-20 中，未标记的部分表示能够满足依赖关系（前一个迭代从正在执行的线程获取数据并结束）。紧邻最右边的条状区域表示一个内核结束框架 n （Z 方向上没有其他未选择的条带）的内核，该进程尚未分配到当前框架可用的 Z 组数据，并且该进程将跳过框架栅栏，即使最后一个内核还没有到达框架栅栏。一旦到达下一框架的第一

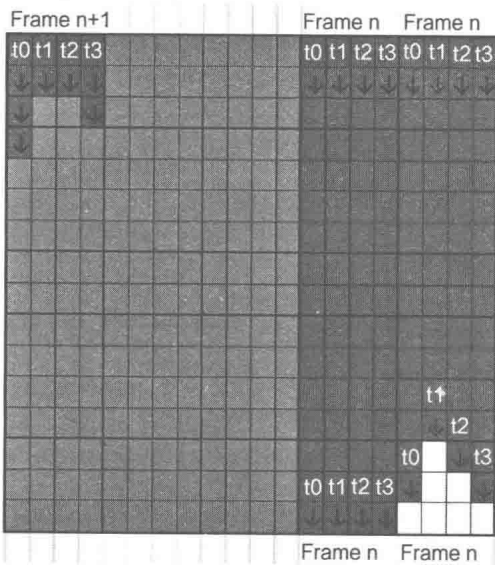


图 5-20 准同步的框架执行过程

列，将会查询第二列框架的执行情况，如果大于或等于下一个 frame-1，则允许该进程进入下一列中。

注意，如果选中的 Z 组和它的邻居位置 Z 的 ($z-1$ 、 z 和 $z+1$) 不是在当前阶段减 1 或之后的位置，那么进程就不能向前继续执行。如果不这么做，向前执行的进程可能会使用到没有及时更新的数据。这就是需要使用 plesiochronous_barrier 的位置。

如果内核在前一个图中没有完成第二列的处理过程，那么处理第一列的内核需要在准同步栅栏中等待。

当内核列（超线程方阵列）的数量超过内核数（方阵的）两倍或两倍以上时，准同步栅栏技术的优势将更加明显。在大规模模型下正是如此（ $512/2=128$ 个方阵列 / 60 核 = 2.1333）。这就表明大模型要比小模型优化效果更好。在观察测试用例结果时，发现任何情况下都能看到主要的优化效果。

图 5-21 展示的是本测试用例的准同步栅栏技术。

```
#if defined(__MIC__)
#define WAIT_A_BIT _mm_delay_32(10)
#else
#define WAIT_A_BIT _mm_pause();
#endif
...
void diffusion_tiled_aligned(
    ... (same as for tiled_HT2)
)

diffusion_tiled(REAL *restrict f1, REAL *restrict f2,
    int nx, int ny, int nz,
    REAL ce, REAL cw, REAL cn, REAL cs, REAL ct,
    REAL cb, REAL cc, REAL dt, int count) {
    // zCountCompleted[nz] is a shared array indicating
    // the iteration counts completed for the z index.
    // Each thread processes all [x,y]'s for given z
    volatile int zCountCompleted[nz];
    for(int i = 0; i < nz; ++i)
        zCountCompleted[i] = -1; // "completed" one before first (0)

    // shared next Phalanx number
    volatile int NextPick = 0;

    // CorePick[nCores] NextPicked'd Phalanx number for core
    volatile int CorePick[nCores];
    for(int i = 0; i < nCores; ++i)
        CorePick[i] = -1; // initialize to less than next pick

#pragma omp parallel
    {
        REAL *f1_t;
        REAL *f2_t;
        int priorCount = -1;
        int myCount = -1;
        int myPick = -1; // (prior pick for 1st iteration of loop)
        // place squads across z dimension
        int nSquadsZ = (nz + nHTs - 1) / nHTs;
        for(;;) {
            if(myHT == 0) {
                // team member 0 picks the next Squad
                CorePick[myCore] =
                    myPick
                    = __sync_fetch_and_add(&NextPick, 1);
            } else {
                // other team members wait until pick made by member 0
                while(CorePick[myCore] == myPick)
                    WAIT_A_BIT;
                myPick = CorePick[myCore]; // pick up new pick
            }
            // ... rest of the loop body
        }
    }
}
```

图 5-21 准同步栅栏 diffusion_tiled_HT3.c 代码摘录


```

} // myHT != 0
myCount = myPick/nSquadsZ; // count interval for myPick
// see if iteration count reached
if(myCount >= count)
    break; // exit for(;;) loop
// determine which buffers are in and out
if(myCount & 1) {
    f1_t = f2;
    f2_t = f1;
} else {
    f1_t = f1;
    f2_t = f2;
}

int z0 = (myPick % nSquadsZ) * nHTs; // home z for HT(0)
int z = z0 + myHT; // z for this team member
int y = 0;

// assure we are within z
if(z < nz) {
    // perform plesiochronous barrier
    priorCount = myCount - 1;
    if(z) // then there is a z-1
        while(zCountCompleted[z-1] < priorCount) // wait z-1
            WAIT_A_BIT;

    while(zCountCompleted[z] < priorCount) // wait z
        WAIT_A_BIT;

    if(z + 1 < nz) // then there is a z+1
        while(zCountCompleted[z+1] < priorCount) // wait z+1
            WAIT_A_BIT;

    int x = 0;
    int c, n, s, b, t; // perform y==0
    y = 0;
    c = x + y * nx + z * nx * ny;
    n = (y == 0) ? c : c - nx;
    s = (y == ny-1) ? c : c + nx;
    b = (z == 0) ? c : c - nx * ny;
    t = (z == nz-1) ? c : c + nx * ny;
    diffusion_tiled_aligned(
        &f2_t[c], // aligned
        &f1_t[c], // aligned
        &f1_t[c-1], // unaligned
        &f1_t[c+1], // unaligned
        &f1_t[s], // aligned
        &f1_t[n], // aligned
        &f1_t[b], // aligned
        &f1_t[t], // aligned
        ce, cw, cn, cs, ct, cb, cc, nx, 1);
    // perform y==1:ny-2
    y = 1;
    c = x + y * nx + z * nx * ny;
    n = (y == 0) ? c : c - nx;
    s = (y == ny-1) ? c : c + nx;
    b = (z == 0) ? c : c - nx * ny;
    t = (z == nz-1) ? c : c + nx * ny;
    diffusion_tiled_aligned(
        &f2_t[c], // aligned
        &f1_t[c], // aligned
        &f1_t[c-1], // unaligned
        &f1_t[c+1], // unaligned
        &f1_t[s], // aligned
        &f1_t[n], // aligned
        &f1_t[b], // aligned
        &f1_t[t], // aligned
        ce, cw, cn, cs, ct, cb, cc, nx, ny-2);

    // perform y==ny-1
    y = ny-1;
    c = x + y * nx + z * nx * ny;
    n = (y == 0) ? c : c - nx;

```

图 5-21 (续)


```

s = (y == ny-1) ? c : c + nx;
b = (z == 0) ? c : c - nx * ny;
t = (z == nz-1) ? c : c + nx * ny;
diffusion_tiled_aligned(
    &f2_t[c],      // aligned
    &f1_t[c],      // aligned
    &f1_t[c-1],    // unaligned
    &f1_t[c+1],    // unaligned
    &f1_t[s],      // aligned
    &f1_t[n],      // aligned
    &f1_t[b],      // aligned
    &f1_t[t],      // aligned
    ce, cw, cn, cs, ct, cb, cc, nx, 1);

// Inform other threads that [z] column is complete
zCountCompleted[z] = myCount;

// perform equivalent of Core barrier
int zEnd = (z0 + nHTs < nz) ? z0 + nHTs : nz;
for(int i = z0; i < zEnd; ++i)
    while(zCountCompleted[i] < myCount)
        WAIT_A_BIT;
} // if(z < nz)
} // for(;;)
} // parallel
return;
}

```

图 5-21 (续)

让我们看看优化后的性能如何。

图 5-22 表示在处理小规模问题时获得了额外的 20% 性能提升。对于解决大规模问题，它又恢复到了与不做修改的 HT1 和 HT2 代码相同的性能。

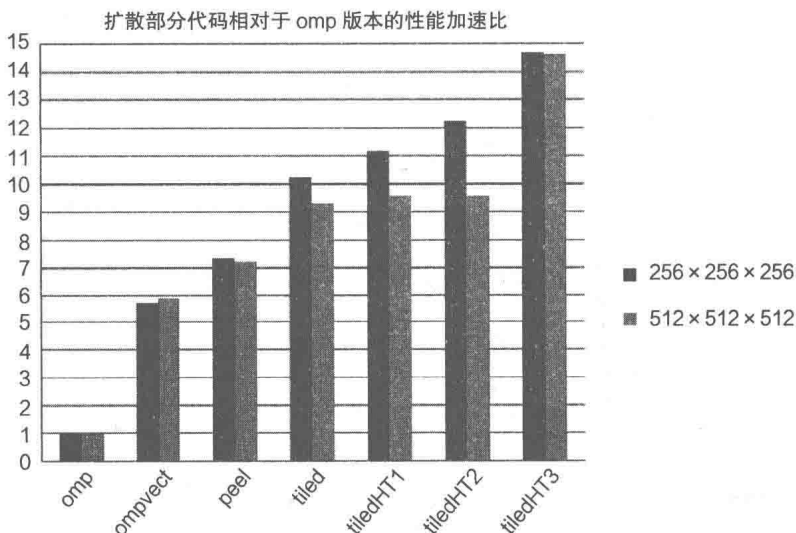


图 5-22 Intel Xeon Phi 5110P 协处理器

在这一点优化上，简化的 OpenMP 版本现在达到了 14 ~ 15 倍的性能提升，比 HPP 书中第 4 章的示例性能提升了 45%。

我们还可以在哪些方面做进一步的优化呢？

5.10 几个留给读者的优化思考

还有一些未在本章中介绍的额外的优化方法，这些可能会在下一步的工作中使用。这些

措施主要包括指导 `prefetch` 和 `clevict` 编译指令，以及插入编译器指令以进行手动代码对齐。当然也可以考虑一下问题规模扩大到单个协处理器内存容量不足以运行程序时可能出现的情况。准同步栅栏技术同样适用于受单个协处理器内存约束的大规模问题。

5.11 类似 Xeon Phi 协处理器的 Xeon 主机性能优化

在这里指出分阶段准同步栅栏技术在多个内核上的性能提升明显是非常重要的。也就是说，甚至在 1P、6 个内核、12 个硬件线程的系统上依然可以高效执行。比如图 5-23 中的 Intel Xeon E5-2620 v2 处理器。

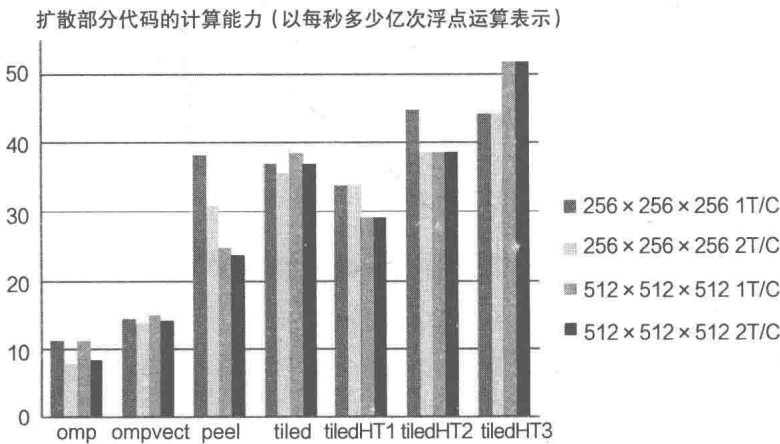


图 5-23 Intel Xeon E5-2620 v2 处理器

注意，图 5-23 展示的主机处理器型号（E5-2620 v2）和之前展示的 Intel Xeon Phi 协处理器型号不相同。这主要是由于主机处理器上的每个内核通常只使用单个线程。然而，我们通常在协处理器上的各个内核核上使用多个线程。通过运行所有相对于 OPenMP 版本的一个或两个线程，很难发现每个内核使用单线程和两个线程的效果差别。因此，我们绘制的图都是使用每秒多少亿次浮点运算的形式表示的。此外，`tiledHT3` 代码总是使用每个内核双线程运行的，因为超线程方阵不能在每个内核上有少于 2 个线程。

通过观察图 5-23 我们发现，在原始“`tiled`”代码运行中每个内核双线程的性能比每个内核单线程的性能还要差。导致这种情况频繁发生的原因是在浮点数密集的程序上每个内核双线程比每个内核单线程表现出更差的性能。显然，准同步超线程方阵的相对执行性能证明了使用每个内核双线程的这个观点。

$$256 \times 256 \times 256 \text{ tiledHT3/tiled} = 1:1.2 \text{ (20\% 的提升)}$$

$$512 \times 512 \times 512 \text{ tiledHT3/tiled} = 1:1.4 \text{ (40\% 的提升)}$$

现在我们将测试在具有 16 核 32 线程的 Intel Xeon E5-2670 处理器的大规模双插槽系统上的性能。

如图 5-24 所示，我们发现了一些意料之外的事情。每个内核双线程的大模型上的 `tiled` 程序获得了不同寻常的性能加速比。在每个内核双线程的大模型上运行其他所有的示例代码（如 `omp`、`ompvect`、`peel`、`tiled`），每个内核双线程的大模型总是运行非常慢。图 5-24 展示了传统的优化技术，不能得出超线程技术是无效的结论。必须运行测试程序才能得到结果。`tiledHT3` 代码的运行结果清楚地说明了使用超线程技术的优势，与分阶段准同步栅栏技术

结合可获得优异的性能。

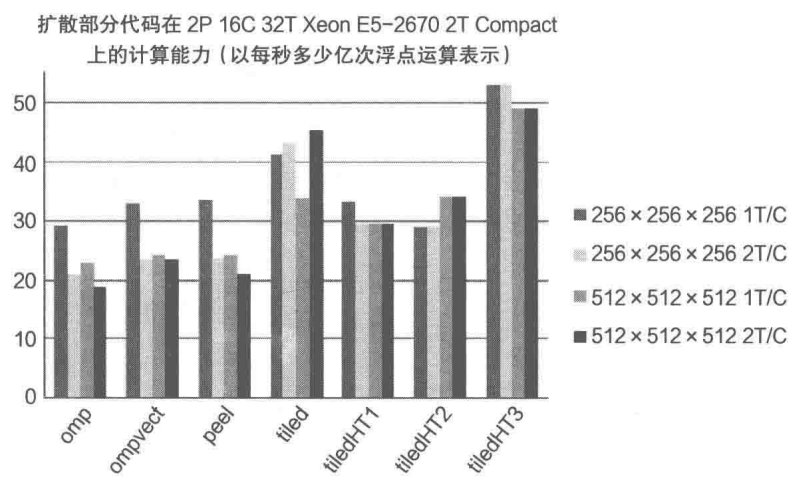


图 5-24 带有 Intel Xeon E5-2620 v2 处理器的双插槽系统

$256 \times 256 \times 256 \text{ tiledHT3/tiled} = 1:2.3$ （23% 的提升）

$512 \times 512 \times 512 \text{ tiledHT3/tiled} = 1:1.086$ （8.6% 的提升）

我们在每个内核单线程上运行大规模 tiled 代码的性能提升百分比如下：

$512 \times 512 \times 512 \text{ tiledHT3/tiled} = 1:1.45$ （45% 的提升）

5.12 总结

本章提出了进一步高效利用全部计算资源的方法。而编译器优化的开发和 #pragma 定向编程不允许在本章的线程调度级上使用，但这并不妨碍我们充分利用平台的计算能力。时间就是金钱。

5.13 更多信息

下面罗列的内容是我们推荐的与本章相关的额外阅读材料：

- http://en.wikipedia.org/wiki/Plesiochronous_system。
- 请在 <http://lotsofcores.com> 上下载本章和其他章节的代码。

故障树表达式并行求解

Jefferson Amstutz

美国，SURVICE Engineering 公司

配置 Intel Xeon 处理器和 Intel Xeon Phi 协处理器的最新硬件支持复杂科学模拟的可视化和交互性。然而，科学和工程模拟中的数据集大小在持续增加，并且期待可以继续维持交互性。处理器和协处理器硬件上的交互性模拟的一个核心组件是开发 SIMD 并行性。本章将会介绍在交互性应用环境中具有大规模输入的故障树表达式求解这一问题中如何发现并利用 SIMD 并行性。

6.1 动机和背景

6.1.1 表达式

表达式通过符号间语法和语义表达数学关系。在本章中，我们关注在运行时经常需要修改的大规模表达式的求值。我们还将关注这些可修改的表达式在表达式规模和实例数量两个方面的扩展。

表达式求值是程序员求解不同计算问题的数学基础。在科学计算和工程应用中，天然地依赖于求解嵌入在模拟程序中的数学表达式。当应用到持续增长的数据集上时，这些表达式的规模非常巨大并十分复杂。通常而言，当已知的模拟表达式可以直接应用到某个程序中时，代码的易用性将牺牲一部分性能。然而，通过开发并行性，并行硬件允许应用程序在大规模数据集求解表达式的同时维护交互性。在某些应用中，可以不同通过编写代码而有效地求解大规模表达式。

6.1.2 表达式选择：故障树

术语“大规模表达式求解”的确切含义是模糊的，并需要指定其合适的使用范围。本章中使用的例子是易损性评估中用到的大规模故障树求解。

易损性评估是表示和量化目标系统中缺陷的过程。这类研究通常属于系统工程。许多系统通过故障树的表示，避免潜在的危险。故障树利用布尔逻辑表达系统间的故障关系。基于组件的可靠性、系统冗余性、物理保护和其他一些潜在的领域相关的参数，模拟方法利用故障树分析（Fault Tree Analysis, FTA）来评估系统的易损性。在 19 世纪 60 年代贝尔实验室首次提出该方法以来，作为分析系统易损性的主要方法 FTA 已经应用在国防、航天、核物理、化学和其他一些高危行业中。

6.1.3 程序实例中的故障树：基本模拟

易损性评估的一个实例是军事应用中的生存性分析。生存性分析是一个非常广泛的研究领域，所以我们只讨论枪击威胁下交通工具的易损性这一特例。射击模拟被广泛应用于评估

交通工具如何抵住不同射击威胁。

现代射击模拟需要许多不同的计算单元来完整模拟不同威胁下的物体易损性。射击时间混乱无序并且十分复杂，这提高了对目标结果量化评估的难度。一种常用的方案是使用致死概率 (P_k) 作为一个指标。 P_k 定义了一个目标系统不再能正常工作的概率。

使用 P_k 指标可在不同的层次上测量易损性。目标整体 (在本例中即为交通工具) 具有几千个组件。因此系统也就由这些独立组件构成。高层次系统定义了交通工具的功能以及最终其在射击危险下的生存性。组件的 P_k 通过工程关系直接影响系统的 P_k 。故障树表达了这些系统 P_k 的关系并且量化了射击事件下的易损性。

理解射击易损性需要尝试百万甚至十亿量级的轨道实例，即可能击中交通工具的威胁。与目标交互的每一个威胁实例都是一个唯一的事件，它需要一个唯一的故障树实例。前面已经介绍过，交通工具故障树包含数千个组件，并且具有复杂的关系。因此代码的可扩展性是故障树求解中维持交互性的关键。交互性允许用户检查目标的生存性结果，这可以促进用户的快速分析和反馈。为了在评估百万 (甚至更高) 量级的大规模表达式时依然可以保持交互性，必须有效地开发并行性。

我们将故障树编译到一种内存编码，支持在每个组件 P_k 值集合上的并行执行。本章中方法主要关注 SIMD 并行性，这是因为应用线程 /MPI 体系结构在不同的领域和实现中并不相同。

注意，我们的实现并不受限于故障树的语义和分析。任意数学表达式可以利用本章讨论的同样原理来最大化 SIMD 效率。射击模拟是一个应用大规模表达式的实例。任意应用一个表达式处理大规模输入的蒙特卡洛模拟都可以从本章的设计方法中受益。

6.2 实例实现

图 6-1 展示了该实例的实现，从输入文件中解析故障树，编译到指令数组，最终使用指令数组求解 N 个组件集的值。

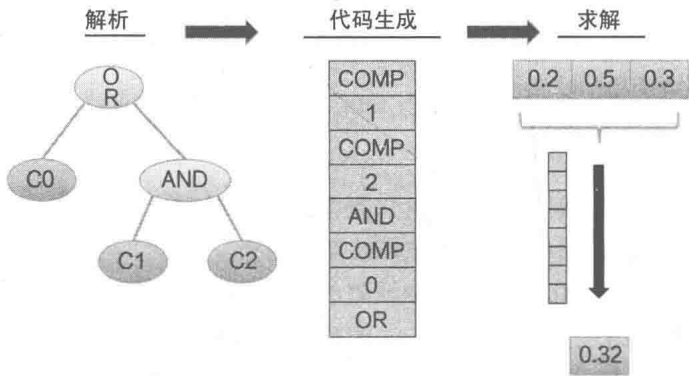


图 6-1 求解故障树的过程

语法和解析结果

故障树实现的输入文件的语法是一个系统定义列表。系统通过包含 “与” 和 “或”(AND/OR) 操作符、常量、组件和子系统的表达式来定义，解析的语法如下：

Expression = ([OR_Operand] | [OR_Operand])⁺
OR_Operand = ([AND_Operand] & [AND_Operand])⁺
AND_Operand = [Name, Constant, Expression]

解析输入是十分简单的，特别是利用已有的工具，例如 yacc 和 lex。然而，输入解析代码的细节超出了本章的范围，并且也与高效求解无关。有兴趣的读者可以分析该实例的输入解析部分。解析输入数据的结果是生成一个解析树，该树包含不同类型的节点，图 6-2 展示了定义

```
struct ftNode
{
    NodeType type; // Enum representing type
                  // (ex: 'op', 'comp', 'end', etc)
    std::string text;
    union {
        struct {
            int index;
        } name;
        struct {
            float value;
        } constant;
        struct {
            ftNode *lhs;
            ftNode *rhs;
        } op;
    };
};
```

图 6-2 节点类包括了节点类型、文件解析文本以及与节点类型相关的信息集

名字节点是输入文件包含的标识符。名字池（即一组组件名字和系统名字）将会在输入解析的同时维护。除非显式地赋予一个表达式，否则名字均为组件（叶节点）。在构造完解析树后，该树将会被遍历并将名字节点根据组件和系统名字池而转变为组件或系统节点。在这一步完成之后，树将被再次遍历并将组件赋予指向输入状态值的索引。一旦组件索引确定，这棵树将会用于评估数组的构造。

构造评估数组

构造一个评估数组或者一个代码数组的目标是建造一组指令集，它可以用来计算给定输入组件值后一个给定系统的值。该实例实现使用反向波兰符号（RPN）来构造系统评估操作。我们选择使用 RPN 是由于以 RPN 形式评估指令的算法十分简洁。图 6-3 展示了一个简单的例子。

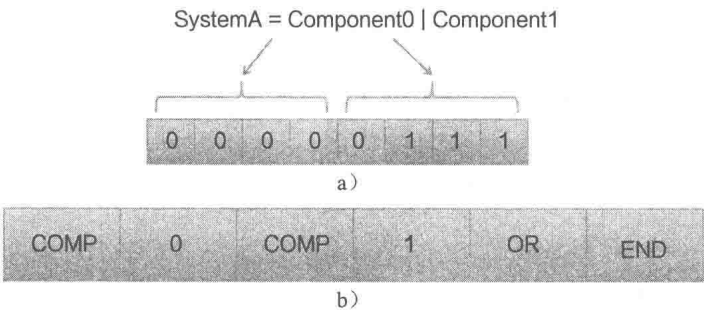


图 6-3 a) 四棵树的输入组件状态：每个元素内的数据表示该组件的索引 b) 系统 A 的 RPN 代码数组

构造代码数组需要遍历解析树。我们使用访问者模式来遍历每个树节点，并逐步增加到评估数组的尾部。求 RPN 表达式的一个需求是每次求解有足够的栈空间。在生成求解代码的同时，最大的栈大小数据会得以维护以保障评估栈的内存空间。更多细节可以参考 <http://lotsofcores.com> 网站上的代码。

需要注意的是，这一部分的计算相对而言不太耗时。在一个可视和交互系统中，代码数

组可以快速修改和重建，在程序中允许可变性将从一个新选择的表达式给出一个立即的反馈。

评估表达式数组

系统评估是我们将要进行优化的主要的计算部分，这是由于这部分计算被嵌入到一个上一层模拟代码中。前面已经介绍，我们的目标是评估 N 棵树，评估系统是共享的。因此，构造的代码数组将会在所有评估实例中共享。系统评估假设 $M*N$ 个组件值已经在上一层模拟中计算完成，这里 M 是系统中待评估的组件数量， N 是将被评估的树的实例数量。图 6-4 中的功能对于每个故障树实例都需要，在每个代码数组的项上循环并根据每个指令的操作符进行计算。

```
void ftEvaluator::evaluateCode(int *code, const int
    numEvaluations, int evaluation, float *compVals,
    float *stack, ftStatus *status)
{
    int sp;           // stack position
    int index;        // system/component index
    float lvalue;     // cache for left value (stack[sp])
    float rvalue;     // cache for right value (stack[sp+1])

    *status = GoodStatus;
    sp = -1;
    int pc = 0;
    for (;;) {
        switch (code[pc++]) {
            case CompType:
                index = code[pc++];
                stack[++sp] = compVals[(numEvaluations *
                                         index) + evaluation];
                break;
            case ConstType:
                // push constant value in next code word on
                // stack as float
                stack[++sp] = *((float *)&code[pc++]);
                break;
            case AndType:
                rvalue = stack[sp--];
                lvalue = stack[sp];
                if (lvalue != 0.0f) {
                    stack[sp] = lvalue * rvalue;
                }
                break;
            case OrType: // assumes statistical independence
                rvalue = stack[sp--];
                lvalue = stack[sp];
                stack[sp] = lvalue + rvalue -
                    (lvalue * rvalue);
                break;
            case EndType:
                // end of code, check stack pointer
                // and return
                if (sp != 0) {
                    *status = StackErrStatus;
                }
                return; // exit for loop
            default:
                *status = BadCodeStatus;
                return; // exit for loop
        }
    }
}
```

图 6-4 代码执行的 C++ 实现：这个版本对于待评估系统的每个评估实例都需要

图 6-4 中的代码在每一个输入组件值上实现了 $AND(P(A)*P(B))$ 和 $OR(P(A)+P(B)-P(A)*P(B))$ 操作。树的每次评估的状态值都保存在一个数组中，以便进行错误检查。在代码数组

执行后，最终的系统值位于每个评估栈的最上端。

使用 ISPC 进行向量化

系统评估的实现十分易于理解，但是并没有利用到 SIMD 执行单元。由于每个故障树评估相同的系统，这属于数据并行操作，因此这是一个应用 SIMD 硬件的完美案例。一个编写显式 SIMD 代码的方式是使用内置函数。虽然这种方式对于我们的示例而言十分简洁，但是在实际代码中这将会给开发人员带来维护负担，他们需要在 SSE4.x、AVX/AVX2、IMCI（在协处理器上）、AVX-512 以及其他更多指令集上进行维护。除了这种费力的维护开销之外，基于内置函数的代码不具有可读性因此难以理解。为了减轻这一问题，我们使用开源的 Intel SPMD 程序编译（ISPC）来实现向量化。图 6-5 中展示的代码实现了前面 C++ 代码的相同功能，但这里利用了 ISPC。

```

inline void evaluateCode(uniform int      *uniform code,
                        uniform int      numEvaluations,
                        varying int      evaluation,
                        uniform float    *compVals,
                        uniform float    *stack,
                        uniform int8     *status)
{
    uniform int sp;          // stack position
    uniform int index;       // system/component index
    varying float lvalue;    // cache for left value
    varying float rvalue;    // cache for right value

    *status = GoodStatus;
    sp = -1;
    uniform int pc = 0;
    while (true) {
        switch (code[pc++]) {
            case CompType:
                index = code[pc++];
                stack[++sp] = compVals[(numEvaluations *
                                         index) + evaluation];
                break;
            case ConstType:
                // push constant value in next code word
                // on stack as float
                stack[++sp] = *((float *)&code[pc++]);
                break;
            case AndType:
                rvalue = stack[sp--];
                lvalue = stack[sp];
                if (lvalue != 0.0f) {
                    stack[sp] = lvalue * rvalue;
                }
                break;
            case OrType: // assumes statistical independence
                rvalue = stack[sp--];
                lvalue = stack[sp];
                stack[sp] = lvalue + rvalue -
                    (lvalue * rvalue);
                break;
            case EndType:
                // end of code, check sp and return
                if (sp != 0) {
                    *status = StackErrStatus;
                }
                return; // exit for loop
            default:
                *status = BadCodeStatus;
                return; // exit for loop
        }
    }
}

```

图 6-5 未优化的 ISPC 评估：这个简单版本直接将 C++ 版本转换为 ISPC


```

    }
}

export void evaluate(const uniform int sectionStart,
                    const uniform int sectionEnd,
                    uniform int numEvals,
                    uniform int compCount,
                    uniform int *uniform code,
                    uniform float *uniform compVals,
                    uniform float *uniform stack,
                    const uniform int stackSize,
                    uniform float *uniform results,
                    uniform int8 *uniform status)
{
    const uniform int end = sectionEnd < numEvals ?
                           sectionEnd : numEvals;
    foreach (i = sectionStart ... end) {
        evaluateCode(code, numEvals, i, compVals,
                    stack + i * stackSize,
                    status + i);

        if (status[i] == GoodStatus) {
            results[i] = stack[i*stackSize];
        }
    }
}

```

图 6-5 (续)

evaluateCode() 函数的这个 ISPC 实现基本上与 C++ 实现一致。唯一的区别是 ISPC 引入的用来指定哪些变量是 ISPC 间统一的声明。除此之外，调用 evaluateCode() 函数的评估函数也在 ISPC 中定义以便利用 ISPC 的 foreach 构件。使用 foreach 代替标准 C/C++ 中的 for 可以使得编译器更好地将循环映射到 SIMD 通道中并行执行。利用这个 ISPC 版本的代码，相对于 C++ 版本，我们可以得到 1.57 倍的加速比。虽然这里得到了性能提升，但是这也表明了没有更好地利用 SIMD。本例中，ISPC 不能完全利用 SIMD 计算能力，这是因为它不能假设 switch 语句始终是一致遍历的。

前两个实现在每个评估的所有的指令中进行迭代。重构这种计算方式，可以在每个指令上迭代计算所有的评估，这可以帮助改进向量化。图 6-6 的代码实现了在每个指令上迭代所有评估。

```

inline void evaluateCode(const uniform size_t secStart,
                        const uniform size_t secEnd,
                        const uniform size_t numEvals,
                        const uniform size_t numComps,
                        const uniform int code[],
                        const uniform int stackSize,
                        uniform float *stack,
                        uniform float *uniform compVals,
                        uniform float *uniform results,
                        uniform int8 *uniform status)
{
    uniform int sp;      // stack position
    uniform int index;   // system/component index
    uniform int pc = 0;  // program counter (code index)
    uniform bool stop;   // flag to break out of eval loop

    uniform float lvalues[SECTION_SIZE];
    uniform float rvalues[SECTION_SIZE];

    sp = -1;
    pc = 0;
}

```

图 6-6 优化的 ISPC 评估


```

stop = false;

// Initialize evaluation status
foreach (i = sectionStart ... sectionEnd) {
    status[i] = GoodStatus;
}

cwhile (true) {
    switch (code[pc++]) {
        case CompType:
            index = code[pc++];
            ++sp;

            foreach (i = sectionStart ... sectionEnd) {
                stack[(stackSize*sp)+i] =
                    compVals[(numEvaluations*index)+i];
            }
            break;
        case ConstType:
            ++sp;
            foreach (i = sectionStart ... sectionEnd) {
                stack[(stackSize*sp)+i] = *((float *)
                                                &code[pc]);
            }
            pc++;
            break;
        case AndType:
            foreach (i = sectionStart ... sectionEnd) {
                rvalues[i-sectionStart] =
                    stack[(stackSize*sp)+i];
                lvalues[i-sectionStart] =
                    stack[(stackSize*(sp-1))+i];
                cif (lvalues[i-sectionStart] != 0.0f) {
                    stack[(stackSize*(sp-1))+i] =
                        lvalues[i-sectionStart] *
                        rvalues[i-sectionStart];
                }
            }
            sp--;
            break;
        case OrType: // assumes statistical independence
            foreach (i = sectionStart ... sectionEnd) {
                rvalues[i-sectionStart] =
                    stack[(stackSize*sp)+i];
                lvalues[i-sectionStart] =
                    stack[(stackSize*(sp-1))+i];
                stack[sp-1] = lvalues[i-sectionStart] +
                    rvalues[i-sectionStart] -
                    lvalues[i-sectionStart] *
                    rvalues[i-sectionStart];
            }
            sp--;
            break;
        case EndType:
            foreach (i = sectionStart ... sectionEnd) {
                cif (sp != 0) {
                    status[i] = StackErrStatus;
                } else {
                    results[i] = stack[0+i];
                }
            }
            stop = true; // exit while loop
            break;
        default:
            foreach (i = sectionStart ... sectionEnd) {
                status[i] = BadCodeStatus;
            }
            stop = true; // exit while loop
            break;
    }
}

```

图 6-6 (续)


```

    }
    cif (stop) break;
}
}
export void evaluate(const uniform size_t sectionStart,
                    const uniform size_t sectionEnd,
                    const uniform size_t numEvals,
                    const uniform size_t compCount,
                    const uniform int code[],
                    uniform float *uniform compVals,
                    uniform float *uniform stack,
                    const uniform int stackSize,
                    uniform float *uniform results,
                    uniform int8 *uniform status)
{
    const uniform int end = sectionEnd < numEvals ?
        sectionEnd : numEvals;
    evaluateCode(sectionStart, end, numEvals, compCount,
        code, stackSize, stack, compVals,
        results, status);
}

```

图 6-6 (续)

注意，图 6-6 中的一般形式是在代码数组中的每个指令的所有评估上使用 ISPC 的 `foreach`（例如，`switch` 语句）。这就需要有一个存储左值和右值的本地缓冲区，而不是使用一个可变的左值和右值变量。缓冲区的大小是段大小，这可以降低数组的空间需求，并且支持调用 `evaluate()` 函数为线程安全的。由于这里的 `foreach` 循环更为紧密，因此 ISPC 能分析出如何更好地进行向量化。

除了通过 ISPC 实现的更好的向量化代码生成之外，我们对初始组件数值使用对齐的内存布局而不是有间隔的布局。这一点在避免 `scatter/gather` 内存访问时特别重要。处理器和协处理器向量读写指令在整个向量可以直接获取和写入时变得更为高效。这种布局通过两种代码特征来实现。首先，我们利用独立数据组件的数组，而不是包含每个数据组件的结构数组（AOS）。如果所有的参数数组在 `evaluate()` 和 `evaluateCode()` 函数中，使用数组结构（SOA）而不是结构数组。除了使用数组结构布局，我们利用组件索引而不是评估索引打包每个元素，从而对齐了组件状态元素。当评估共享相同代码数组的每个实例集时，所有单独的组件索引将会被访问。内存中连续的组件索引值避免了 `gather` 操作，如图 6-7 所示。

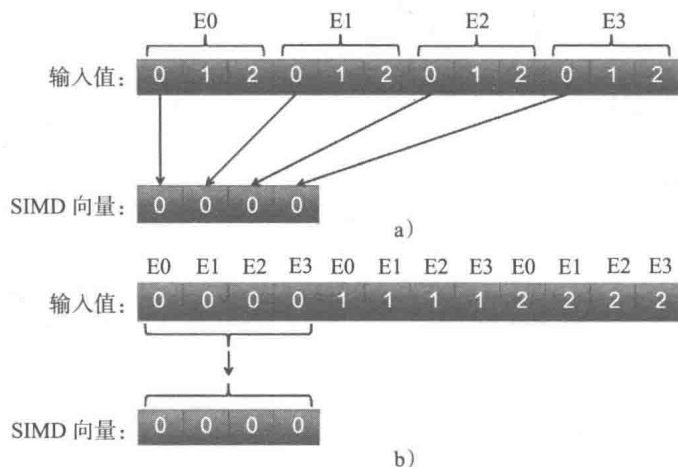


图 6-7 内存布局: a) 间隔: 在评估时需要重组元素, 导致一个 `gather` 操作, 它需要四次访问操作。b) 对齐: 元素通过组件索引组织, 允许执行一个向量读取操作

相对于第一个 ISPC 实现, 第二个 ISPC 版本实现获得了 7.9 倍的加速比, 相对于 C++ 实现, 快了 12.7 倍。这里我们利用单线程来测量性能, 以便显示 SIMD 的性能提升。因此, 这个加速比是更优秀的 SIMD 可扩展的结果 (也来自于更好的缓存访问)。评估树是一种易并行操作, 因此, 实现一个多线程的解决方案十分简单。

6.3 其他因素

科学和工程应用根据不同的体系结构和设计而不同, 一些程序可以被优化到更高的性能。在这个射击模拟实例中, 故障树整体并没有始终用于评估。在实践中, 一个输入文件包含交通工具的所有系统和组件关系, 但是用户经常会选择一个没有包括所有组件的系统。通过利用一个压缩的组件集, 可以在评估故障树时节约可观的内存容量。这可以通过在处理最初组件数值时, 从全局组件索引 (所有组件) 到压缩组件索引 (与特定系统相关的组件) 的转化来实现。在进行评估之前执行这个转变操作可以节约评估时每个组件查找的索引转变开销。处理输入表达式的子集的应用程序利用这种内存压缩方式将会得到性能提升。本章的示例代码使用这种优化。

另一个优化是组织计算为大规模多线程程序。多核处理器和众核协处理器允许多线程程序独立运行。然而, 每个线程至少要处理 SIMD 指令宽度大小的数据子集。这样在包含 16 位宽的浮点向量的 Intel Xeon Phi 协处理器上的一个评估包至少包含 16 个评估, 并且需要保持前一节讨论的数据布局。执行评估包的线程可以独立地到达系统评估阶段并保持最大的 SIMD 效率。

另一个值得注意的是, 本章中讨论的 SIMD 优化技术可以应用到任意 SIMD 硬件体系结构中, 例如 GPU 和 ARM NEON CPU。本章中的射击模拟代码首先在开源的可视化模拟实验室 (VSL) 框架中利用 NVIDIA 的 CUDA GPU 环境实现。

6.4 总结

处理器和协处理器上的 SIMD 硬件发展支持许多科学和工程应用达到交互性的能力。这些应用中的大规模表达式也需要 SIMD 并行化。蒙特卡洛模拟是可以有效利用 SIMD 硬件并行的一类模拟。我们在本章中学习了作为射击模拟中大规模表达式的故障树求解可以通过 ISPC 有效地利用 SIMD。

我们的示例实现解析了输入故障树, 生成了评估指令数组, 并利用 SIMD 并行性执行这些数组。我们考虑了最终评估代码中可以改进计算性能的两个特征。首先, 重组 ISPC 代码以利用更小的循环, 这可以帮助 ISPC 编译器更好地生成向量化代码。其次, 利用对齐的数组结构数据布局方式可以减少向量的 scatter/gather 访存操作。

6.5 更多信息

与本章内容相关的更多的阅读材料如下:

- 一般故障树分析信息: en.wikipedia.org/wiki/Fault_tree_analysis。
- 利用 GPU 的复杂故障树分析: Aghassi,H.,Aghassi,F,2013.Fault tree analysis speedup with gpu parallel computing.Int. J. Comput. Inf. Syst. Indust. Manag. Appl. 5,106-114。
- 有关 ISPC 的更多信息可参考 ispc.github.io。
- VSL 由美国 Army Reaserch Laboratory 开发。更多项目信息和代码可参考 vissimlab.org。
- 本章和其他章节的代码可在从 lotsofcores.com 网址下载。

深度学习的数值优化

Rob Farber

美国, TechEnablement.com

随着大规模并行协处理器普遍应用于视频、音频以及社交媒体等海量数据,人们对深度学习数值优化算法的研究热情正在复苏。各种实际应用的经济价值是使用这些算法的驱动力,这些应用包括更新或更强大的互联网搜索、基于复杂实时模式认知的无人驾驶汽车以及增强现实 workflows 等。在生产环境的实时机器学习和数值优化问题上,使用右侧并行映射,单一 Intel Xeon Phi 协处理器在持续混合型单双精度性能表现上可以超过每秒亿万次浮点运算 (TF/s)。十分有趣的是,在不依赖于某些优化库的情况下 (比如 Intel 数学内核库),此种性能表现也是可以在用户自行开发的代码上实现的。由此证明了非 Intel 之流的“普通”程序员经过代码编译也可以写出高性能的应用程序。通过利用消息传递接口协调数千 Intel 协处理器的计算,可以达到近线性可扩展和千万亿次的计算性能。例如本章将讨论的大规模并行映射及相关代码在平均持续混合精度训练中可以达到每秒 2.2 千万亿次 (PF/s) 计算。该实验是在 TACC (德州高级计算中心) Stampede 超级计算机提供的 3000 个 intel Xeon Phi 协处理器上进行的。高性能以及大容量内存是 Intel Xeon Phi 协处理器家族的突出优点。Intel Xeon Phi 协处理器家族因此成为海量数据训练方面的理想平台,尤其可以用于解决大数据相关的复杂问题以及多维度模式识别。

7.1 拟合目标函数

数学建模是应用数学分支之一,它对现实进行抽象,可用于分析和预测。数据驱动的数值模型一般可以通过计算机实现,从而使它可以成为有用的分析和嵌入式系统工具。具体地,通过暗含神经网络推断的函数有预测功能,意味着它可以在一个时间序列中正确地预测未来值,执行分类任务,处理信号,以及对机器人以及实时系统中的复杂和不可预测的外界刺激的应对和适应。具体请参考 Duda 和 Hart, 以及 Farber 和 Lapedes 发表的相关文章,这些文章已在 7.8 节注明。

数据驱动模型拟合可以理解为函数优化的一种形式,它对一系列模型参数进行调整,从而令该模型以最小的误差程度适用于某些数据集合。上面的误差由一个目标函数所确定。该函数有时称为代价函数,可用来评估对于给定的一系列参数,模型与数据的适应程度 (图 7-1)。

$$\text{Error} = \text{func}(P_0, P_1, \dots, P_n)$$

图 7-1 目标函数

最小二乘法 (LMS) 的目标函数经常用于将一组的 N 个观测数据点拟合为一个模型,该模型常由曲线或曲面来表示。图 7-2 中定义的最小二乘法误差总是正数,这意味着一个最佳拟合误差应该为 0。由于数值精度问题、噪声数据和其他挑战性问题,参数化模型很少实现零误差。通常,用于确定模型参数的数值优化技术会求出局部最小值,或者求出代价函数的

低点，而优化算法总也绝不会脱离最小值和最低点。最佳拟合或者全局最小值不是一定可以求出的，但是最小二乘法目标函数的误差值可以作为衡量模型和数据匹配程度的手段。

机器学习和数值优化技术在各行各业以及科研中的广泛使用表明，由于数据量的大小、复杂度、噪声和其他甚至涉及局部最小化因素的困扰，基于经验确定参数的模型（如线性回归、非线性回归、神经网络等）往往能提供“足够好”的解决方案，有时甚至是唯一的可行方法。

基于人工神经网络（ANN）的训练可以表示为一个函数优化问题，它尝试确定将训练数据集误差最小化的最优网络参数（例如，模型参数或内部网络的权值和偏差）。训练过程中计算代价是高昂的，因为需要使用不同的参数集合来重复计算目标函数。由于数据集中每个样本的误差都需要计算，所以每个目标函数评价的运行时间为 $O(N_{\text{param}} \times N_{\text{data}})$ 。在大多数情况下，相对于训练数据的数量 N_{data} ，参数的数量 N_{param} 是较小的，从而意味着总体的运行时间主要是由训练数据集来决定的。实际上，数据的严格规范化相对于参数数量而言是更重要的，所以 ANN 是不记忆训练数据集的。这就是为什么交叉验证数据集（例如，包含在训练期间未呈现给神经网络的数据的独特例子的数据集）显得如此重要，因为交叉验证可以检测 ANN 可以很好地拟合训练数据但是概括不全的情况。

数据并行目标函数的并行化通过大规模并行计算可以显著减少运行时间，详见图 7-3，图中公式表明了随着处理器数量的增多运行时间会缩短，这称为强可扩展，因为对于给定的数据集，运行时间随处理元素个数的增加而成比例减少。

$$\text{Error} = \sum_i^N (\text{Known}_i - \text{Predicted}_i)^2$$

图 7-2 误差差值平方和

$$O\left(\frac{N_{\text{data}} * N_{\text{param}}}{N_{\text{processors}}}\right)$$

图 7-3 一个数据并行目标函数理想的运行时间

大多数数据并行目标函数是浮点精度密集型并适于向量化的，这些特性使得数据并行目标函数在所有内核及协处理器的每个单核向量单元上都可以有效率地运行。可扩展的并行性与高效的向量化，表明了数据密集型机器学习和数值优化问题可以很好地适应 Intel Xeon Phi 协处理家族（例如图 7-4 的右下方）的高性能结构。

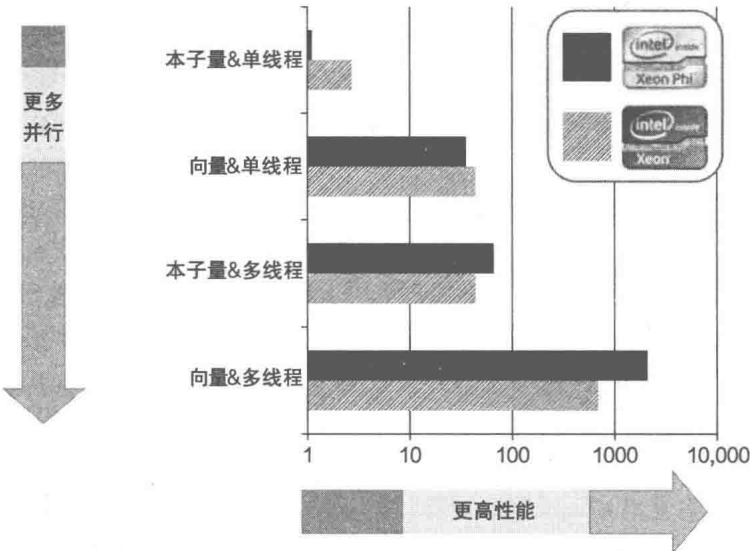


图 7-4 Intel Xeon Phi 协处理器性能概要

(图片由 James Reinders Intel 公司提供)

本章中的神经网络训练代码将会使用我研发的大规模并行映射，是在 20 世纪 80 年代在美国洛斯阿拉莫斯国家实验室和圣菲研究所研发的。Thearling 调研了各种并行化神经网络映射，其中包含我的“Faber”映射（调研的链接见 7.8 节）。

- 不可行实现：神经网络中的每个节点都映射到一个处理器，但对于大规模并行，这个方法会受限于内存带宽和网络通信消耗。
- 改进方法：张某提出了一种改进方法，他和他的同事仔细地将多重网络节点映射到同一个处理器。这种方法可以减少通信消耗并能够更加高效地计算网络节点的输出。
- 除了其他大规模并行系统之外，Faber 映射还可以在单个协处理器和协处理器集合上有效率地运行，如图 7-5 所示。根据 Thearling 的调研结果，这个方法是迄今为止可以达到的最快性能。自从 20 世纪 90 年代，该方法已经成为了世界范围内的一种常用方法。

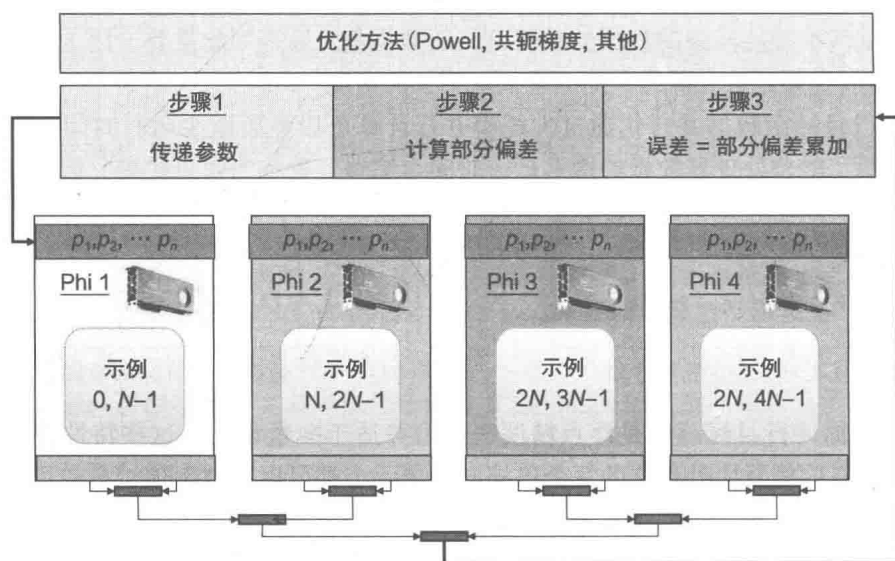


图 7-5 一般映射：能量 = func (P1, P2, ... Pn)

图 7-5 中的标有“最优化方法”文本框表明许多通用的、基于库的数值优化库可以通过大规模并行映射利用。《Numerical Recipes》这本书中涵盖了多种数值优化算法，是一本值得推荐的学习资料。

本章中的所有结果都是使用当前较为流行的并且是免费可用的开源 nlopt 优化库来完成的。nlopt 优化库是 Johanson 写的，可以通过网络下载，下载地址为 <http://ab-initio.mit.edu/nlopt>。当然，读者也可以使用众多免费数字许可工具包代替 nlopt，包括 SLATEC、NAG（数值算法组）、MINPACK、GNU 的科学库、Matlab、Octave、scipy、gnuplot、SAS、Maple、Mathematica 等。

在图 7-6 中展示了 Faber 映射在 TACC Stampede 超级计算机提供的 3000 个协处理器上的近线性扩展表现。使用 TACC Ranger 超级计算机提供的 60000 个处理器内核，美国 ORNL（橡树岭国家实验室）Titan 超级计算机提供的 16384 个 GPU，以及 Thinking Machines CM-2 上的 64000 个处理器内核，连同 20 世纪 80 年代以来的许多其他超级计算机和集群，也可以得到类似的构图——它们的共性是用眼睛看上去是线性的。

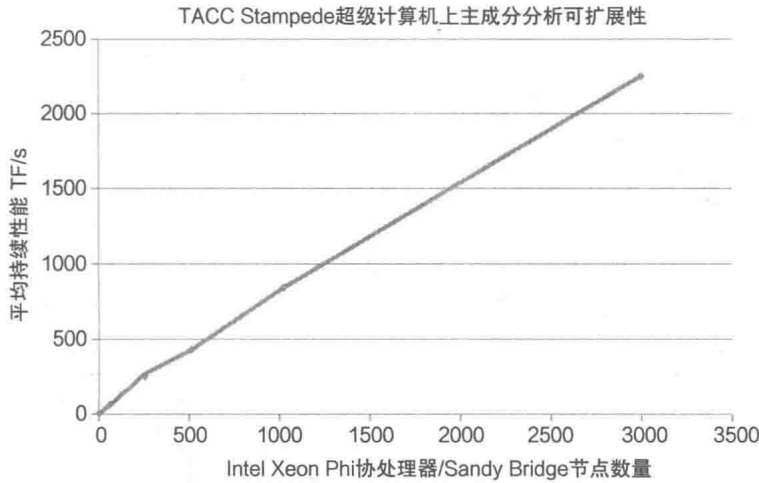


图 7-6 在 TACC Stampede 超级计算机上扩展到 3000 个 Intel Xeon Phi 协处理器

以上的近线性扩展的映射是以下 1 ~ 3 步的运行结果：

- 第 1 步：把参数广播到所有处理器。通常，MPI 的全局广播是非常高效的，并且具有恒定的运行时间开销。广播数据的时间开销通常受限于网络互联硬件的带宽。
- 第 2 步：每个 MPI 节点独立计算存于本地内存中的数据子集的偏差，这意味着随着处理节点的数量变化，运行时间可以表现出很强的可扩展性。本步骤的运行时间是由每个 MPI 节点处理的数据量以及单节点的计算速度决定的。（注：每次计算开始时，每个 MPI 节点都会加载部分数据集，而且一般会根据可用网络带宽进行扩展。）
- 第 3 步：在超大型系统上，Faber 映射的可扩展性主要受归约运算的运行时行为所影响。大多数 MPI 归约运算需要的运行时间为 $O(\log_2(N_{\text{nodes}}))$ ，其中 N_{nodes} 为 MPI 节点数量。实验中，网络互连的延迟会限制归约运算的性能，由于部分和中的字节数量是很小的，因此通常每 64 位的部分和会有 8 字节。

经验表明，即使是最大规模的问题也可以利用单精度（32 位）浮点格式来存储参数和数据，但是在上面第 2 步和第 3 步中累积的偏差需要利用双精度（64 位）来进行存储运算。否则，在使用单精度类型存储运算偏差时，数百万次偏差累加会造成精度损失，由此造成的信息丢失可能会导致优化算法陷于局部极小值。使用 64 位双精度累加器来实现混合精度归约运算造成的运行时间影响是最小的。

近线性扩展性只能通过基于有单一误差值而非误差向量的目标函数的优化技术来实现。一些优化算法会利用一个误差向量来确定下一组参数，从而评价整个优化过程。此种方法的缺点是误差向量的大小需要根据数据量的大小进行扩展。在多对一的通信模式下运行优化算法时，即使是非常小的误差向量也会很快占满 MPI 节点的网络间连接或者子系统内存和处理器。因此，只有并行运行并且消除了误差向量尺度相关问题的基于误差向量的优化算法才能使用此种映射。

7.2 目标函数与主成分分析

神经网络是一种通用的计算方法，理论上它可以学习任何可计算函数。支持上面观点的理由很简单，因为它可以实现所有的逻辑门（例如，or、xor、and、not 等），而恰恰这些逻

辑门用于构建通用计算设备，比如通用计算机。

最近，神经网络已经成为通过训练多层神经网络的各个层次进行深度学习的一种流行方式。通过定义隐层，人工神经网络可以模拟其他计算设备，创建复杂的数据内部表示，或者实现特征提取。特征提取指的是数据突出特性或者属性的识别，以便于在后续的任务中用它来进行回归或者分类。Lapedes 和我发现只有两个隐层需要执行建模、预测和符号学习任务。尽管有两个以上的隐藏层，但是聪明的人类辅助设计可以创建利用更少参数的更小的网络，或者用微调来使复杂的深度学习任务完成得更好，详见 Hinton 的报告。

本章将主要讨论使用多层 ANN 来实现主成分分析 (PCA) 以及非线性主成分分析 (NPCA)。

PCA 广泛应用于数据挖掘和数据分析来减少数据集的维度和提取数据集的特征。主成分分析可以通过使用一组正交直线来表示一个数据集的最大方差，其中每条直线都由观测变量的加权线性组合所定义。同样地，非线性主成分分析可以利用连续开放或者封闭的曲线来表示数据方差。圆即是一个自我连接且没有终点的封闭曲线的例子。

ANN 的体系结构可以通过数量受限的线性隐藏神经元或者瓶颈神经元 (在图 7-7 标记为 B) 来寻找数据集的主成分，如图 7-7 所示。类似地，当一个非线性算子用于瓶颈神经元时，图 7-7 中的 ANN 可以用来实现非线性主成分分析。这些非线性算子既可以是开放曲线，也可以是闭合曲线。

非线性主成分分析在模式识别、生物建模、天气建模和化学等众多具有挑战性问题的领域都有广泛应用。目前有大量的网站、教程和书籍都是关于非线性主成分分析的。当然，直接搜索“nlpca”将会是一个很好的学习起点。

在训练期间，在图 7-7 中的多层神经网络通过输入训练集中的每个训练向量来呈现，即图底部的 I 神经元。实质上，ANN 通过线性或非线性的主成分进行自我学习。ANN 使输入向量信息通过底部一半神经网络 (从而将每个输入向量从高维空间降为低维空间，例如从二维降为图 7-7 中的一维)，并通过顶部一半神经网络在输出神经元上重建原始的高维输入向量，从而使误差达到最小化。这是一个非监督学习的例子，因为目标函数只基于训练集中的输入数据计算误差。(特殊地，LMS 误差可以通过对输入及输出神经元的差值进行减法及平方运算求得。) 另外，当向输出神经元提供标准输出结果时，即为监督学习。该标准输出结果反映了对每一个训练样例的期望输出或已知输出，这样 LMS 误差可以通过对标准输出结果及输出神经元的差值进行减法及平方运算求得。

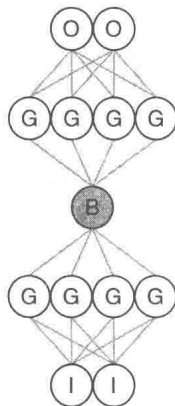


图 7-7 一种多层的 PCA 或者 NLP 神经网络

7.3 软件及样例数据

本章中 PCA 以及 PLPCA 样例的完整代码以及编译脚本可在 github 上的 farbopt 代码仓库中找到 (参考 7.8 节中的 ULR)。

图 7-8 中的代码段描述了 OpenMP 实现的 func() 函数，该函数是用于训练神经网络的 LMS 目标函数，可运行在单一设备或跨越全部计算节点的 MPI 环境中。可以看出，代码实现比较容易。不考虑运行环境 (例如，单一协处理器或者 MPI 实现版本)，以下代码执行了大部分的计算工作。


```
{
    {
        err=0.; // initialize error here for offload
#pragma omp parallel for reduction(+ : err)
        for(int i=0; i < nDeviceExamples; i++) {
            float d=myFunc(i, param, deviceExample,
                           nDeviceExamples, NULL);

            err += d*d;
        }
    }
}
```

图 7-8 一种 OpenMP 版本的 LMS 数据并行目标函数

用户提供的内联函数 `myFunc()` 用来计算对于训练数据集中特定样例的模型误差。OpenMP 编译器负责将归约循环在多个处理器内核上正确地进行并行化。程序员负责描述 `myFunc()` 函数，这样编译器会对 `myFunc()` 并行实例正确地向量化，以充分利用协处理器中每个内核的向量单元。

注意，`func()` 或者 `myFunc()` 的调用是通用的，即 `farbopt` 样例代码可以用来优化任意可并行化的目标函数，例如支持向量机、独立成分分析、期望最大化以及其他目标函数。

本章中用到的样例代码实现了一个 C 语言版本的人工神经网络，如图 7-7 所示，该网络会通过 LMS 进行训练。通过条件编译，可在编译时为 B 及 G 神经元的瓶颈指定转换类型。可选项包括针对 PCA 分析的线性操作符以及几种针对 NLPCA 分析的 sigmoid 函数实现。通过使用条件编译，首先可以简化测试及基准化分析，其次可以使编译器更好地优化 `myFunc()` 内联函数，以充分利用协处理器上的向量单元。

在 ANN 参数（例如权重以及神经元偏差）的优化中，LMS 目标函数通过 `nlopt` 优化库调用。实际 `nlopt` 函数调用显示在下面的代码段中。`getTime()` 函数返回总体优化进程的时间。在 `func()` 函数中类似的测试装置会记录目标函数的时间开销，通过进一步验证，用户可以得知 LMS 目标函数确实决定了应用程序整体的运行时间。

MPI 代码采用“主/从”执行模式，其中优化方法在主计算节点上运行，而工作负载被划分到从计算节点上运行。为了提高执行效率，主计算节点也可以作为从节点对目标函数进行评估。

在 MPI 从节点中，MPI 编号大于 0 的节点在调用 `MPI_Init()` 函数后调用 `startClient()` 函数。`startClient()` 的代码如图 7-9 所示。

```
void startClient(void * restrict my_func_data)
{
    int op;

    double xFromMPI[N_PARAM];
    double partialError,sum;

    for(;;) {
        // loop until the master says I am done - then exit
    }
```

图 7-9 MPI 从节点代码


```

MPI_Bcast(&op, 1, MPI_INT, 0, MPI_COMM_WORLD);
// receive the op code
if(op==0) { // we are done, normal exit
    break;
}
// receive the parameters
MPI_Bcast(xFromMPI, N_PARAM, MPI_DOUBLE, 0,
          MPI_COMM_WORLD);

partialError =
    objFunc(N_PARAM, xFromMPI, NULL, my_func_data);
MPI_Reduce(&partialError, &sum, 1, MPI_DOUBLE,
          MPI_SUM, 0, MPI_COMM_WORLD);
}
}

```

图 7-9 (续)

优化方法（一个 NLOPT 样例如图 7-10 所示）通过 `mpiObjFunc()` 实现，该函数首先运行在主节点上，然后执行 MPI 通信，也执行部分目标函数。代码如图 7-11 所示。

```

double startTime=getTime();
// initialize error here for offload
int ret=nlopt_optimize(opt, x, &minf);
printf("Optimization Time %g\n",getTime()-startTime);

```

图 7-10 通过 NLOPT 调用启动优化进程

```

double mpiObjFunc(unsigned n, const double * restrict x,
double * restrict grad,
void * restrict my_func_data)
{
    int op;
    double partialError, totalError=0.;
    // Send the master op code
    MPI_Bcast(&masterOP, 1, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Bcast((void*) x, N_PARAM, MPI_DOUBLE, 0,
              MPI_COMM_WORLD); // Send the parameters

    partialError = objFunc(N_PARAM, x, NULL,
                          my_func_data);
    MPI_Reduce(&partialError, &totalError, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD); // get the totalError

    return(totalError);
}

```

图 7-11 MPI 主节点的目标函数代码

7.4 训练数据

通过实验对 ANN 的 PCA 版本（采用线性操作符编译）的可扩展性及运行时间进行测试，测试数据集包含对围绕一条直线呈雪茄形分布的噪声数据的 3000 万个观测值，噪声数据的平均值为 0，方差为 0.1。对该分布的小规模抽样如图 7-12 所示。

类似地，通过实验对 NLPKA 版本的 ANN 的运行时间进行测试，测试数据为从如图 7-13 所定义的 z_1 与 z_2 的非线性关系中提取的 3000 万个点。为了使 ANN 的训练过程更有难度，再次引入均值为 0 且方差为 0.1 的噪声数据。

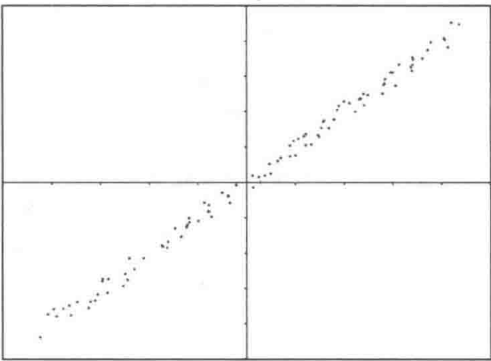


图 7-12 PCA 的线性数据集样例

$$\begin{aligned} z_1 &= t + e_1 \\ z_2 &= t^3 + e_2 \end{aligned}$$

图 7-13 一种非线性关系

一个简单的 sigmoid 函数 $x/(1+|x|)$ 的代码实现如图 7-14 所示，该函数用于对如图 7-7 所示的 ANN 中的 G 及 B 神经元进行非线性转换。sigmoid 函数的主要优势是我们可以知道确切的浮点运算数量。尽管这个简单的 sigmoid 函数可用于基准测试，但更好的 sigmoid 函数（例如 $\tanh()$ ）以及逻辑函数通常应用在实际应用中，这是由于训练过程中的参数优化可以更快地收敛到一个好的结果。

```
char *desc="Eliott activation: x/(1+fabsf(x))";
inline float G(float x) { return( x/(1.f+fabsf(x)) ) ;}
```

图 7-14 一个简单的 sigmoid 函数

图 7-15 展示了 NLPKA 数据以及 ANN 模型预测结果。

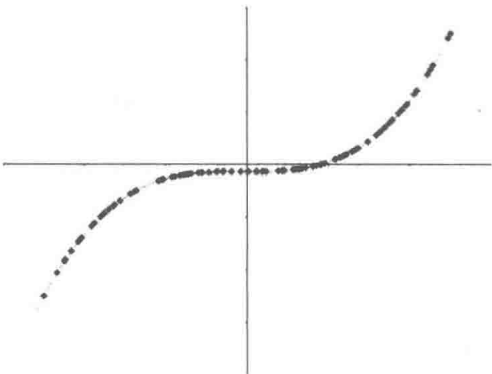


图 7-15 NLPKA 数据样例以及 ANN 预测

7.5 运行时间

结果样例程序的输出结果验证了 LMS 目标函数的运行时间实际上决定了应用程序的运行时间（如图 7-16 所示）。

	模式	数据大小	nparam	占funco的时间百分比 (%)
PCA	Native	30MB	83	99.1324
NLPCA	Native	30MB	83	99.7844
PCA	Offload	30MB	83	99.9998
NLPCA	Offload	30MB	83	99.9992

图 7-16 本地以及卸载模式下 PCA 以及 NLPCA 在 func() 函数中的运行时间

图 7-17 展示的性能结果可以证明，当以本地执行模型运行时，单一协处理器执行性能可以超过每秒 10^{12} 次浮点运算 TFLOPS。即使最高浮点运算速度及最低浮点运算速度之间的差值很大，其平均性能十分接近所观测到的最高性能每秒 10^9 次浮点运算 (GFLOPS)。

	模式	最小值(GF/s)	平均值(GF/s)	最大值(GF/s)
PCA	Native	15.1494	1012.8	1037.03
NLPCA	Native	22.0354	369.315	374.58
PCA	Offload	14.2368	946.761	973.886
NLPCA	Offload	20.5346	359.672	363.216

图 7-17 两个输入的 PCA 以及 NLPCA ANN 的浮点运算性能

在卸载执行模式中，即使有将参数移动到协处理器以及每次调用 func() 函数时的偏差检索等开销，还是展现出很好的浮点运算性能。该卸载执行模式下浮点运算性能与预计达到的性能一致，其中包含了当单一计算节点中运行多个协处理器或者通过 MPI 在多节点集群或顶级的超级计算机中运行时产生的通信开销。当运行多个设备时，无论协处理器通过本地模式执行 MPI 程序还是通过卸载模式由主处理器间接调用，通信开销是无法避免的。在 TACC Stampede 超级计算机上的运行结果如图 7-4 所示，其中单一协处理器通过卸载模式被主处理器上的 MPI 程序所访问。

farbopt github 代码库中还包含了一个基于 Python 的神经网络生成器，命名为 genFunc.py。该生成器可以用来对不同 ANN 架构的性能影响进行调研。例如，为了计算相对于如图 7-7 所示架构加载的两个向量长度的误差，一个拥有 4 个输入神经元的神经网络需要从内存中载入 4 个浮点数据值。通过图 7-17 及图 7-18 中的性能结果对比可以看出额外数据加载带来的性能影响。

	模式	最小值(GF/s)	平均值(GF/s)	最大值(GF/s)
PCA	Native	23.8488	822.883	855.79
NLPCA	Native	31.0903	430.143	437.162
PCA	Offload	22.2739	821.65	850.331
NLPCA	Offload	27.3932	427.277	434.41

图 7-18 四个输入的 PCA 以及 NLPCA ANN 的浮点运算性能

特别要注意的是，由于需要读取 4 个数据值而不是两个数据值所导致的内存带宽限制，PCA 的性能会下降。与此相反，NLPCA 计算的计算频度更高，这意味着它会花费更多的时间处理向量寄存器，不是等待从内存中检索的数据。在卸载模式及本地模式下，拥有 4 个输入神经元的神经网络性能均会优于拥有两个输入神经元的较小神经网络，因此可以说，NLPCA 的性能受益于较大神经网络带来的额外参数。

7.6 扩展结果

在单一协处理器上基于训练数据集规模的可扩展性如图 7-19 所示，并表明本地模式下的 PCA 代码在几个较大训练数据集上都获得了超过每秒 10^{12} 次浮点运算的性能。此外，主处理器（12 核 3.3 GHz 的 Intel Xeon X5680 处理器）的性能相对于协处理器的性能会迅速地以每秒 10^9 次浮点运算级速率呈阶梯形显著降低。图中的平线表明处理器受到资源的限制。协处理器近似达到每秒 10^{12} 次浮点运算的渐近性表现也表明在资源方面受到限制。图中所显示的最高性能接近处理器单精度运算峰值的 50%。实验结果表明内存带宽很可能是限制两个设备性能的罪魁祸首。

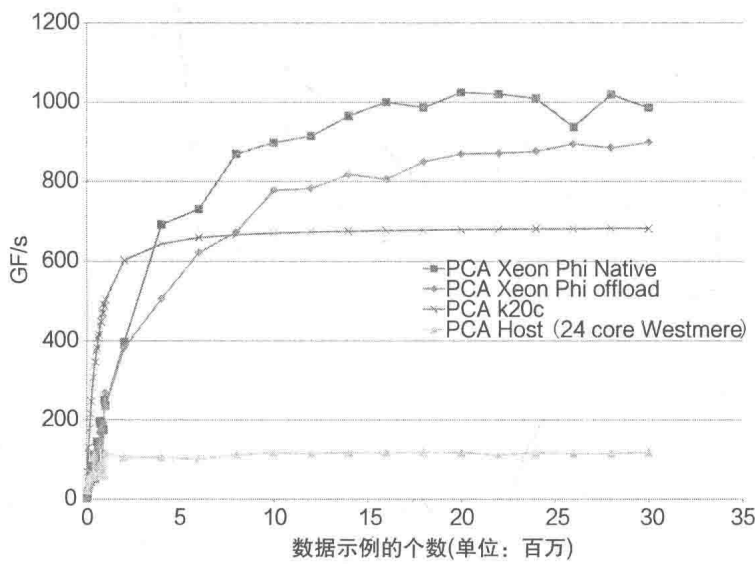


图 7-19 数据集规模可扩展性测试

正如前面所讨论的，在 TACC Stampede 超级计算机上，在 3000 个协处理器上通过 MPI 并行训练一个 ANN 可以到达接近线性的可扩展性及每秒 10^{15} 次浮点运算 (PFLOPS) 的性能 (如图 7-3 所示)。可以预测在将来拥有数万协处理器的系统中，同样会展现出接近线性的可扩展性以及每秒百亿亿次浮点运算的性能。

7.7 总结

从单设备每秒 10^{12} 次浮点运算性能，到实际观测中的每秒 10^5 次浮点运算性能，再到未来每秒 10^{18} 次浮点运算性能，Intel Xeon Phi 产品系列为研究及商业应用打开了一个新前景，并将处理器的性价比及性能功耗比提升到一个新级别。X86 处理器以及 Intel Xeon Phi 协处理器达到高性能的共同关键因素包括以下两方面：多处理器间接近线性的可扩展性；每个设备中运行由大规模并行映射产生的繁重的向量化处理负载。例如，在基于 x86 架构的 TACC Ranger 超级计算机上，这些相同的应用样例可以扩展到 60 000 个处理内核上，并获得每个时钟周期 4 个单精度操作的实际性能。

尽管训练过程是计算密集型的，但是本章阐述的内容证明该训练过程可以很好地映射到大规模并行系统中，并且最终的参数化模型在多种低功耗以及小内存的设备上快速运行，这些设备包括 CPU、DSP、FPGA 甚至 ASIC 硬件。ANN 的上述特征及其学习执行复杂模式认

知、信号处理、分类任务的能力，从一定程度上解释了为什么 ANN 会在大规模互联网搜索引擎、对处理时间高要求的机器人以及实时视觉应用等应用领域得到广泛应用。

通过接触与使用本章阐述的概念以及样例程序，读者可以开始探讨与理解前馈神经网络以及递归神经网络的学习能力。递归神经网络是一种利用网络架构中前向以及后向连接的 ANN。通过在 ANN 中加入有向环，递归神经网络可以创建和使用内部存储器来解决问题和处理任意输入序列。递归神经网络提供了非常高的计算数据比（即协处理器平均每字节数据传输上需要执行大量的浮点算术操作），这是因为在神经网络中需要进行正向与反向的迭代信息传输。因此，递归神经网络非常适合众核协处理器加速。

除此之外，通过众核协处理器加速，研究人员可以解决多目标优化带来的计算挑战。这些目标函数在科学研究、工程实践以及日常生活中都普遍存在，但是很难解决。例如，大部分的消费决策都需要对成本与效益进行权衡。类似地，工程师利用经常会面对承重与强度设计带来的挑战。对于多目标函数，权衡多种相互冲突的目标意味着许多可能的解决方案需要调研，因为每一项相对于其他项的加权重要性在不断变化（例如，某些时候强度比承重更重要，反之亦然）。这些优化的时间开销很大，且受限于局部最优解。然而，Intel Xeon Phi 协处理器以及其他大规模并行部件提供了每秒 10^{12} 次浮点运算的性能甚至每秒 10^{15} 次浮点运算的性能，研究人员可以利用这些计算资源去深入研究这些常见且重要的但计算量大的优化问题。

7.8 更多信息

- Duda,R.,Hart,P. E.,Stork,D.G.,2001. Pattern Classification,second ed. Wiley.
- Farber,R.,2011. CUDA Application Design and Development. Morgan Kaufmann.
- Lapedes,A.S.,Farber,R.,1987. How neural networks work. Proceeding of IEEE Denver-Conference on Neural Netorks. Denver: IEEE.
- NLPCA:www.cs.toronto.edu/~hinton/ and <http://nlpca.org>.
- Johnson,S.G.,2014. The NLOpt nonlinear-optimization package. <http://ab-initio.mit.edu/nlopt>.
- github 上的 farbopty 库, <https://github.com/rmfarber/farbopt>.
- 开源 nlopt 优化库,<http://ab-initio.mit.edu/nlopt>.Thearling, Massively Parallel Architectures and Algorithms for Time Series Analysis,<http://www.thearling.com/text/csss93.htm> 或者 Addison-Wesley 1993 *Lectures in Complex Systems*.

优化聚集 / 分散模式

Simon J. Pennycook*, Christopher J. Hughes†, Mikhail Smelyanskiy†

* 英国, Intel 公司; † 美国, Intel 公司

为了在节能的方式下拥有高计算能力, 很多现代微处理器依赖于单指令多数据流 (SIMD) 的执行方式。这些微处理器 (包括被最新的 Intel Xeon 处理器和 Intel Xeon Phi 协处理器所使用的微处理器) 是主要为数据在内存中连续排列所优化的。不过, 这些微处理器通过聚集和分散操作, 使得数据在内存中不连续存放的时候, 也能支持 SIMD 的执行方式。

如何执行聚集和分散操作是特定于平台的, 有时是在软件中发生的, 有时利用专门的硬件指令。但是它们的功能总是一样的: 一个聚集操作从内存中读取一些地址不连续的数据, 然后把它们装进单个 SIMD 寄存器; 一个分散操作从一个 SIMD 寄存器中取出数据, 然后把它们写回各自独立非连续的存储单元。涉及聚集 / 分散操作的存储地址, 直到运行之前是不知道的, 并且可以表示任何存取模式 (包括连续存取), 允许程序员 (和编译器) 对于即使是最不规则的循环体也能向量化执行。

图 8-1 是一个包含直接读和直接写并需要聚集 / 分散操作来向量化的循环体示例。当我们向量化循环体时, 单个 SIMD 迭代会读一组 $A[B[i]]$ 并写一组 $D[E[i]]$ 。一般而言, $A[B[i]]$ (和 $D[E[i]]$) 在执行之前都在未知的存储位置, 也可能存储在非连续的地址里。因此, 通常的 SIMD 加载和存储指令是不能满足需要的, 那些指令只能获取一组连续的数据。因而, 我们需要一个聚集操作来读取地址非连续的 $A[B[i]]$ 数据和一个分散操作来写数据到 $D[E[i]]$ 。

```
for (i = 0; i < N; i++) {
    C[i] = A[B[i]];           // indirect read
    D[E[i]] = C[i] + some_constant; // indirect write
}
```

图 8-1 一个引起聚集和分散的循环体的示例

聚集和分散相对于连续的 SIMD 加载和存储, 会要求硬件做更多的工作——它们通常会有更高的指令开销, 更小的可预测性, 更可能需要更多的高速缓存行或者页 (这取决于特定的访存模式)。编程者因此应尽可能地通过避免间接和非连续的获取数据, 来最大化的减少它们的使用。

然而, 间接性可能是一个算法的固有部分。例如, 数据元素可能按一个输入依赖顺序存取, 或者为了降低计算复杂性, 计算会在数据元素的子集上执行。这些算法很常见, 尤其是在一些数据元素之间的关系事先不知道的结构域里 (例如: 图表问题、非结构网络), 而且替代算法也同样未必能执行——即使它们的 SIMD 效率很高。

非连续的存取也是程序员常选择的数据布局, 一个好例子是把数据另存为结构数组 (AoS) 而不是数组结构 (SoA)。图 8-2 表明了一个数组的两种数据布局, 这个数组有 4 个元素, 每个元素由三个结构体成员 $\{x, y, z\}$ 组成。

AoS 使得程序员可以根据数据类型来编写应用程序，这些数据类型在它们的领域是有意义的（例如：粒度分布），但是并不适合在 SIMD 上执行，为多重数据元素获取单一结构体成员都不会是连续的。SoA 在某种程度上可以连续获取，但不是在所有情况下，只有在数组本身不是通过间接存取的情况下，才能保证连续的加载和存储。更进一步来讲，使用 SOA，单一元素的结构成员间的距离会导致较差的缓存行为。即使是混合算法（例如：数组结构数组（Array-of-Structure-of-Arrays））也无望成为在所有情况下最好的布局设计。更糟的是，对于同一段代码中不同的部分需要不同的布局来达到最优化，而且外部库通常需要的数据是在特定形式下的（这超出了应用开发者所能控制的范围）。简而言之，为应用选择最佳数据布局依赖于很多因素，SIMD 的使用只是众多因素中的一个。

当必须使用聚集和分散时，或者当它们太容易避免时，帮助硬件使得它们尽可能快很重要——它们的性能在某种程度上取决于程序员控制下的一些事情。本章剩下的部分详细说明了一系列在处理器和协处理器上优化聚集和分散模式的方法：特别地使用域的知识来提高时间和空间的局部性；选择一个适当的数据布局；在 AoS 和 SoA 之间执行即时置换；通过多重循环迭代来隐藏聚集和分散的开销。

我们把这些方法应用于 miniMD benchmark 的聚集和分散模式上，miniMD benchmark 来自于美国圣地亚国家实验室的 Mantevo 套件（Sandia National Laboratories' Mantevo Suite），使用了伦纳德（Lennard-Jones）内部原子势。然而，我们讨论的优化方法适用于广泛的应用。

8.1 聚集 / 分散在 Intel 架构下的说明

完全明白这里的优化方法需要一些关于 Intel 指令集的知识。尤其重要的是对于聚集和分散操作的支持。8.5 节有指令集文档的链接。

对于程序员来说，使用 Intel SIMD 流指令扩展（Streaming SIMD Extension, SSE）和 Intel 高级矢量扩展（Advanced Vector Extension, AVX），聚集和分散操作都需要通过标量加载和存储以及 shuffle 指令执行。标量加载和存储读写独立的数据元素，shuffle（例如，pinsrd、extractps、vinsertf128）把独立的元素或元素组插入或提取出向量寄存器。

AVX2 包括几种不同类型的聚集指令。这些聚集指令采用基址寻址方式，把一组 32 位或者 64 位有符号整数的索引装进一个 SIMD 寄存器，同时把一个结束掩码装进一个 SIMD 寄存器的符号位——设定其屏蔽位的数据是聚集在一起的；其他数据没有聚集在一起，而且参与计算的目的数据是没有修改的。这些指令在结束后会清除掩码。

在最新一代 Intel Xeon Phi 协处理器上使用的 Intel 初级多核指令（IMCI）同时包括聚集指令和分散指令。这里的聚集指令和 AVX2 的聚集相似，但是使用了一个屏蔽寄存器来存储结束掩码。而且，聚集指令只读取存储在一个（通常有 64 个字节宽）缓存行内的数据。由于聚集的数据可能被多个缓存行分散存储，因此软件通常通过验证结束掩码来判断所有的数据是否已经读取，如果没有，跳转回循环的开头并重新执行指令。因此，一个聚集操作的性能会随着存储器中被聚集数据的物理距离的增长而降低。分散指令是类似的，相对于聚集指

结构数组



数组结构

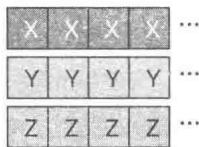


图 8-2 高性能计算应用程序的两种最常见数据布局

令从存储器中读取数据，分散指令把源 SIMD 寄存器中的数据写到存储器中。

最后，AVX-512 包括的聚集和分散指令同 IMCI 的聚集和分散指令非常相似，但是如同 AVX2 的聚集指令，它们在一条执行指令中读、写全部数据，而不考虑需要使用多少缓存行。AVX-512 将会被下一代的 Intel Phi 协处理器和被 Intel 命名的代号为 Knights Landing 的处理器所支持。

注意 SSE 为英特尔架构引进了浮点型 SIMD 功能，以及用于操作 128 位向量寄存器的指令。新的指令被引入到 SSE2、SSE3 和 SSE4 中，但 SIMD 的宽度仍是固定的，直到 AVX 引入 256 位的指令。类似地，AVX2 引入了新的指令，而没有更改 SIMD 的宽度。IMCI 把 SIMD 的宽度增长到 512 位，但是有特别针对于最新一代协处理器的指令；AVX-512 也把 SIMD 的宽度增长到 512 位，但是另外也会支持 SSE 和 AVX 指令集。

8.2 聚集 / 分散模式在分子动力学中的应用

分子动力学是一种 N 体模拟的示例，分子动力学应用以一种相对简单的方式模拟大量原子或分子（即，体）在空间中可能的运动轨迹——每一个原子受到其他原子施加的合力，因此其速度和位置都会被影响。

性能分析 miniMD，我们选择的分子动力学代码（如：Intel VTune 放大器），强调两个关键研究热点。第一个关键研究热点是力计算 kernel，它占了将近 80% 的运行时间。力计算 kernel 是计算原子间作用力的 kernel。力的基本计算复杂度为 $O(N^2)$ ，由于每一个原子对所有其他的原子都有直接或间接的作用力。这对于大量的原子来说是相当高的开销。因此，分子动力学典型封装把每一个原子上的作用力拆分成两个典型的组成部分：短程力，计算所有间距小于 R_c 的原子对间的作用力；长程力，估算不需要直接计算的所有原子对间的作用力（如：通过快速傅里叶变换）。这种对于作用力的拆分使得力的计算复杂度降为 $O(Nk)$ ， k 是每一个原子邻居个数的期望值（它本身是一个关于模拟参数的函数，如密度和 R_c ）。作为基准测试代码，miniMD 没有包括长程力的估算 kernel——此后我们的关注点都是近程力。第二个关键研究热点是构建邻居列表 kernel，它占了 10% 的运行时间，该 kernel 构建的列表中所包含的原子对是间距小于截断距离与一些边缘（skin）距离 R_s 之和的原子对。这个边缘距离对于一些时间步长来说允许列表重用，以考虑比实际需要更多的原子对作为代价。

图 8-3 表明了一个二维图像上的原子邻域——在三维空间中，邻域是一个球体，模拟区域是一个立方体。模拟空间也分成“格”（cell）或者“箱”（bin），它们用来加速构建邻居列表。通过使用这种方法谨慎地离散化空间，并在每个箱中预先计算出一个原子列表，构建邻居列表也可以避免 $O(N^2)$ 的复杂度——对于每一个原子来说，这仅对于核对预确定的一组邻居节点附近的 bin 来说是必要的，而不是全部的模拟区域。

这两个关键研究热点最原始的 miniMD 源代码在图 8-4 和图 8-5 中给出，即分别是短程力的计算和构建邻居列表 kernel。如这些图所示，对于每一个 kernel 来说，都有两个要使用聚集 / 分散操作的原因；相关行用加粗字体来强调。第一，为了找到每一个原子索引（ i ），邻居原子索引（ j ）存储在查找数组（`neighs` 和 `loc_bin`）中。结果是邻居位置必须是聚集的（从 x 轴来看），而且对于邻居原子的作用力的计算贡献必须累积进力数组（`f`），方法是执行一个聚集操作一个减法运算，最后，一个分散操作。从代码中除去这种间接操作只在一种情况下是可能的，即通过引进冗余计算和存储——由于不同原子的邻域会部分地不可预测地重叠，无法排序原子使得所有（最小限度的）邻域通过连续的索引来顺序执行。第二，位置和力数组

存储在 AoS 中，连同结构的尺寸——由编译时定义的变量 (PAD) 所控制。然而，把这些数组转换成 SoA 不会影响代码中的聚集 / 分散操作的数量，由于原子索引本身是不连续的。

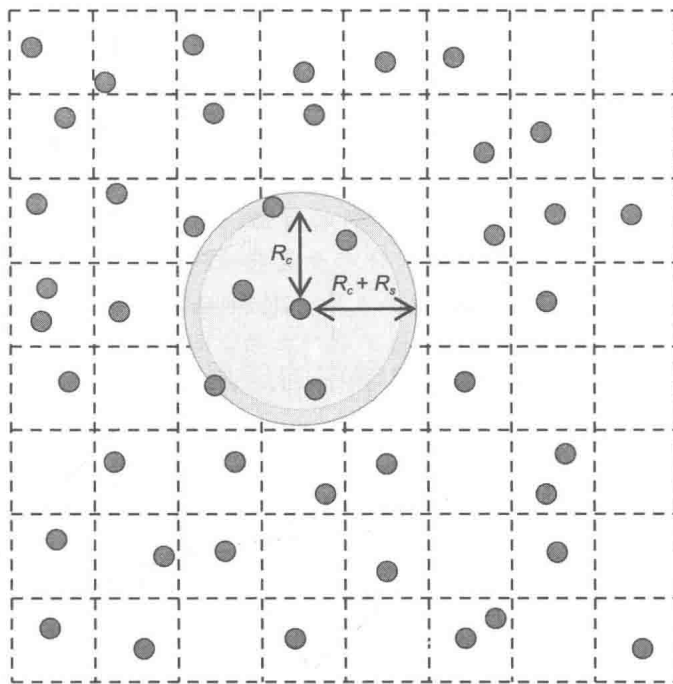


图 8-3 一个粒子在二维空间中的邻域，它由 R_c 截断距离和 R_s 边缘距离所定义

```

for (int i = 0; i < nlocal; i++) {
    neighs = &neighbor.neighbors[i*neighbor.maxneighs];
    const int numneighs = neighbor.numneigh[i];
    const float xtmp = x[i*PAD+0];
    const float ytmp = x[i*PAD+1];
    const float ztmp = x[i*PAD+2];
    float fix = 0.0; float fiy = 0.0; float fiz = 0.0;
    #pragma simd reduction(+:fix, fiy, fiz)
    for (int k = 0; k < numneighs; k++) {
        const int j = neighs[k];
        const float delx = xtmp - x[j*PAD+0];
        const float dely = ytmp - x[j*PAD+1];
        const float delz = ztmp - x[j*PAD+2];
        const float rsq = delx*delx + dely*dely + delz*delz;
        if (rsq < cutforcesq) {
            const float sr2 = 1.0 / rsq;
            const float sr6 = sr2 * sr2 * sr2 * sigma6;
            const float force = 48.0*sr6*(sr6-0.5)*sr2*epsilon;
            fix += delx * force;
            fiy += dely * force;
            fiz += delz * force;
            if (GHOST NEWTON || j < nlocal) {
                f[j*PAD+0] -= delx * force;
                f[j*PAD+1] -= dely * force;
                f[j*PAD+2] -= delz * force;
            }
            if (EVFLAG) {
                // compute contribution to energy/virial
            }
        }
        f[i*PAD+0] += fix;
        f[i*PAD+1] += fiy;
        f[i*PAD+2] += fiz;
    }
}

```

图 8-4 miniMD 上最初的短程力的计算循环 (为了节省空间省去了能量 / 维里的计算)


```

for (int i = 0; i < nlocal; i++) {
    int* neighptr = &neighbors[i * maxneighs];
    int n = 0;
    const float xtmp = x[i*PAD+0];
    const float ytmp = x[i*PAD+1];
    const float ztmp = x[i*PAD+2];
    const int ibin = coord2bin(xtmp, ytmp, ztmp);
    for (int k = 0; k < nstencil; k++) {
        const int jbin = ibin + stencil[k];
        int* loc_bin = &bins[jbin * atoms_per_bin];
        if (ibin == jbin)
            // check for neighbors in own bin
        else {
            for (int m = 0; m < bincount[jbin]; m++) {
                const int j = loc_bin[m];
                if (halfneigh && !ghost_newton && (j < i))
                    continue;
                const float delx = xtmp - x[j*PAD+0];
                const float dely = ytmp - x[j*PAD+1];
                const float delz = ztmp - x[j*PAD+2];
                const float rsq =
                    delx*delx + dely*dely + delz*delz;
                if ((rsq <= cutneighsq)) neighptr[n++] = j;
            }
        }
        numneigh[i] = n;
    }
}

```

图 8-5 miniMD 上最初的邻居列表的构建循环 (为了节省空间省去了查看自己箱里的邻居节点)

其他的分子动力学代码 (如: LAMMPS, 也是由美国桑迪亚国家实验室研发的) 可能会有额外的聚集 / 分散操作出现在这些循环的内部。在以不同种类的原子为主的模拟中 (如: 氢和氧), 代码段中的一些常量 (如: 截断距离) 实际上是不同的, 这基于考虑范围内的原子种类的结合。此外, 更复杂的是, 原子间势能 (如: 内嵌的原子方法) 基于两个原子间的计算距离 (因为直接计算力值的开销太大), 会引入多余的查表找。

考虑到在分子动力学模拟过程中计算力所占的高时间比, 而且计算力要求很多聚集分散操作, 那么减少聚集 / 分散操作的开销会显著提高整个应用的性能。

8.3 优化聚集 / 分散模式

8.3.1 提高时间和空间的局部性

存储系统对于独立数据的读写 (它们构成聚集 / 分散操作) 的响应对于操作的性能会有很大的影响。通过有效的运作, 写延迟可以隐藏, 而需要用到读结果的指令要等到读完成才能执行——这导致了聚集比分散对于延迟更敏感。如果任意操作的数据不在缓存内, 或者数据分散在很多缓存行里 (即: 这里几乎没有重用 / 时间局部性), 那么聚集和分散操作的完成会变得更慢 (即: 这里没有多少空间局部性)。对于分子动力学示例来说, 一个能同时提升聚集 / 分散操作的时间和空间局部性的方法是排序。如我们下面将要描述的一样, 排序不需要是确切的, 即使是局部排序也能提高时间或者空间的局部性, 因此, 这是一个提升性能的好机会。

为了提高时间局部性, 我们可以通过原子的空间位置来给它们排序。如果原子在空间上相邻, 那么它们就很有可能拥有相同的邻居。最外面循环的 (原子之上的) 邻近迭代会使用聚集 / 分散操作, 因此, 这个排序会使得该操作命中很多相同的邻居。对于一个通过聚集或者分散操作访问的给定邻居, 这被缓存命中的可能性会很大, 由于这个邻居有可能是其他最

近获取的原子的邻居。

为了提高每一个独立的聚集 / 分散操作的空间局部性，我们可以在每一个原子的邻居列表里排序索引。这将极大地减少一个已知的聚集 / 分散操作访问的缓存行数量。

对于所有应用来说，排序可能不是提升空间或者时间局部性最好的方式，但是，每一部分数据的重用率最大化和每一个聚集 / 分散操作访问最少的缓存行的原则，是普遍适用的。

图 8-6 中的图比较了原子在如下两种情况下的执行时间：随机排序和通过空间位置排序。该排序使用了简单的扫描线排序，基于原子的箱索引——在最新版本的 miniMD 中这是一个默认方法，并且在 LAMMPS 中也有模仿的排序。两个平台对于随机排序都占用了大量的计算资源，在处理器上，排序对于性能有了 1.48 倍的提升，在协处理器上，排序对于性能提升了 1.36 倍。

在之前的工作中，我们已经证明了通过多线程的空间分解（随着在每一个线程的本地域内的空间排序）可以得到更高级别的时间和空间局部性，因此获得了更大的性能提升。我们仍然在同 miniMD 最初的开发者讨论最好的方法来整合这种分解到最初版本的基准测试中——这一章的所有结果都使用所谓的“原子分解”，每一个线程只简单地分配相同数量的原子。使用原子分解能提供更好的负载均衡，但是一个线程可能会需要访问属于其他线程的原子，这种可能性会随着线程数量的增长而增加。

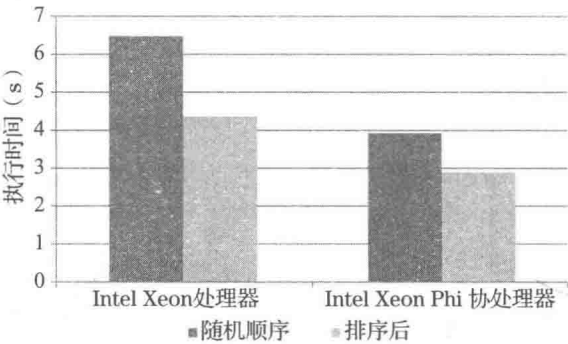


图 8-6 比较原子随机排序和原子基于空间位置排序（扫描线排序）的执行时间

8.3.2 选择一种适当的数据布局：AoS 与 SoA

如之前讨论过的，选择 AoS 或者 SoA 作为数据布局是根据数据的访问模式来决定的——根据经验法则：如果数据是通过间接元素访问的并且从一个结构需要访问多个元素，就使用 AoS；如果不是，则使用 SoA。

为了更好地理解这种经验法则，考虑最好和最坏情况下两种数据结构的聚集 / 分散操作的存储访问模式，假设 SIMD 的宽度是 N ，一个拥有 M 个元素的结构体，一个长度（按数据个数来算）为 C 的缓存行，如表 8-1 和表 8-2 所示。对于这两种数据布局来说，最好的情况是聚集和分散操作的数据索引是连续的；最坏的情况是聚集和分散操作的数据索引是不连续的。需要注意的是，在最坏的情况下两个因素在实际情况中所占的比例，不考虑对齐问题，访问的数据可能会分散在两个缓存行里。

表 8-1 当从一个长度为 M 的结构中加载 1 个数据，AoS 和 SoA 访问的缓存行的数量

数据布局	最好情况（连续）	最坏情况（不连续）
AoS	$\text{ceil}(N \times M/C)$	N
SoA	1	N

表 8-2 当从一个长度为 M 的结构中加载 M 个数据，AoS 和 SOA 访问的缓存行的数量

数据布局	最好情况（连续）	最坏情况（不连续）
AoS	$\text{ceil}(N \times M / C)$	$2 \times N \times \text{ceil}(M / C)$
SoA	M	$2 \times N \times M$

为了把这些等式应用到 miniMD 上，在协处理器上单精度运行，我们设 $N = 16$ ， $M = 3$ ， $C = 16$ （由于一个 64 个字节的缓存行能存储 16 个 4 字节的浮点数）。由于我们在三维空间上获取一个原子的位置和力，因此我们关心的是第二组等式。由于每一个原子的邻居是不连续的，因此如果我们使用 SoA 而不是 AoS 作为数据布局方式，我们希望每次聚集或者分散操作可以命中三倍以上的缓存行。通过设 $M=4$ （即：为结构体添加一个额外的元素），我们能保证对齐并且通过两个因素中的一个来减少最坏情况下的缓存行数量。

miniMD 已经使用了 AoS，因此不能期望转换成 SoA 可提高性能。确实，在实验中，我们在两个平台上观察到了很小的速度降低（<5%）。性能的差别如此之小反映了 miniMD（排序后的）并不是存储限度——聚集 / 分散操作的指令开销才是性能的瓶颈。如之前说明的，从 AoS 或者 SoA 加载 $\{x,y,z\}$ 数据都是聚集操作，因此编译器产生的指令序列在两种情况下都是一样的（最重要的区别是布局对地址计算的作用）。

8.3.3 AoS 和 SoA 之间的动态转换

前一节的结果并不意味着 AoS 和 SoA 通常是可以互换的，并且没有任何性能影响。事实上，由于存储访问模式的不同，不同的优化技术适用于两种不同的数据布局。在 miniMD 中从 SoA 上收集位置和力的 $\{x,y,z\}$ 数据总是需要三个聚集操作：一个用来收集 x 的值，一个用来收集 y 的值，一个用来收集 z 的值。另一个方面，从 AoS 收集可以当成一个单独的收集结构，紧随其后的是两者之间的布局转换。

当视为转换操作时，很明显，有些操作会引入冗余。例如：硬件每次可以从缓存中加载结构的一个元素，因此需要三次访问来读完一个结构。如果我们不是在一次访问中读取一整个结构，我们就需要减少访问的次数和请求指令的数量。代价是我们必须使用一系列额外“shuffle”指令重新排序 x 、 y 和 z 值。

图 8-7 解释说明了对于 miniMD 位置数组使用 SSE 的部分 AoS-to-SoA 转换过程。我们一次加载一个完整的邻居（即：4 个连续的 32 位值，包括填充）到 SIMD 寄存器里。在该例子里，加载的邻居是带索引 i 、 jk 和 l 的。然后使用连续的 7 个 shuffle 指令或置换指令来实现转换（一个构建如图所示的四个中间寄存器，另一个构建最终的每个 x 、 y 、 z 的寄存器）。为了借助更大的 SIMD 来扩展指令集的序列长度，每个向量寄存器中需要加载多个原子。实际上，我们要把一个 256 位的 AVX 寄存器当成两个 128 位的通道，把一个 512 位的 IMCI 寄存器当成 4 个 128 位的通道。

SoA-to-AoS 转换过程就是 AoS-to-SoA 的逆过程，使用相同的 shuffle 序列。除了存有 SoA 格式 x 、 y 、 z 值向量寄存器外，我们还要在一个寄存器中存储填充的值（通常是 0）。

图 8-8 和图 8-9 摘录了用来完成这些转换的 AVX-512 内置函数代码。在单精度的情况下完整的转换内置函数代码，以及其他指令集的代码，作为源代码的一部分可以在本书配套的网站（lotsofcores.com）上下载。

位置数据

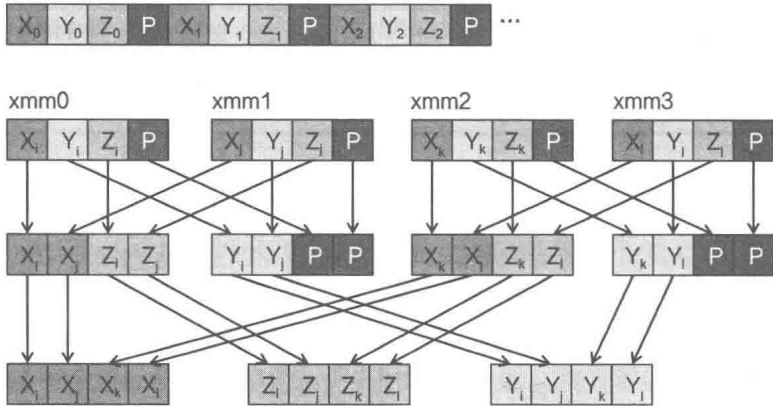


图 8-7 在 AoS-to-SoA 转换过程中寄存器和存储的内容。字母 P 对应的数据是填充的，用来保证数据的对齐

```
#define _MM_BCAST4_PS(a) _mm512_extload_ps(a,  
_MM_UPCONV_PS_NONE, _MM_BROADCAST_4X16, _MM_HINT_NONE)  
#define _MM_MASK_BCAST4_PS(v, m, a)  
_mm512_mask_extload_ps(v, m, a, _MM_UPCONV_PS_NONE,  
_MM_BROADCAST_4X16, _MM_HINT_NONE)  
  
_mm512 tmp0 = _MM_BCAST4_PS(&data[indices[0]]);  
tmp0 = _MM_MASK_BCAST4_PS(tmp0, mask_00F0,  
&data[indices[4]]);  
tmp0 = _MM_MASK_BCAST4_PS(tmp0, mask_0F00,  
&data[indices[8]]);  
tmp0 = _MM_MASK_BCAST4_PS(tmp0, mask_F000,  
&data[indices[12]]);  
  
// repeat loads for tmp1 with indices[1,5,9,13]  
// repeat loads for tmp2 with indices[2,6,10,14]  
// repeat loads for tmp3 with indices[3,7,11,15]  
  
_mm512 xz01 = _mm512_mask_swizzle_ps(tmp0, mask_0xAAAA,  
tmp1, _MM_SWIZ_REG_CDAB);  
_mm512 yw01 = _mm512_mask_swizzle_ps(tmp1, mask_0x5555,  
tmp0, _MM_SWIZ_REG_CDAB);  
_mm512 xz23 = _mm512_mask_swizzle_ps(tmp2, mask_0xAAAA,  
tmp3, _MM_SWIZ_REG_CDAB);  
_mm512 yw23 = _mm512_mask_swizzle_ps(tmp3, mask_0x5555,  
tmp2, _MM_SWIZ_REG_CDAB);  
  
x = _mm512_mask_swizzle_ps(xz01, mask_0xCCCC, xz23,  
_MM_SWIZ_REG_BADC);  
y = _mm512_mask_swizzle_ps(yw01, mask_0xCCCC, yw23,  
_MM_SWIZ_REG_BADC);  
z = _mm512_mask_swizzle_ps(xz23, mask_0x3333, xz01,  
_MM_SWIZ_REG_BADC);
```

图 8-8 单精度数据的 AoS-to-SoA 转换的节选代码，使用的是 IMCI 内置函数

注意 不熟悉编译器内置函数和 Intel 指令集的读者可以阅读 Intel 内置函数手册和指令集参考手册来帮助理解如下所示的节选代码，这些资源的链接在本章的结尾可以找到。

表 8-3 和 8-4 比较了默认编译器产生的聚集 / 分散操作的指令数，以及使用了之前提到的优化方法后的聚集 / 分散操作的指令数。需要注意的是，512 位指令的数量是在最坏的情况

下，假设 3 个聚集操作访问了 16 个缓存行并且每个缓存行使用了 3 条指令（vgatherps、cmp、jmp）。即将推出的下一代处理器和协处理器将会支持 AVX-512 的指令，这项计数将会下降到三条指令。对于每个使用 AVX-512 的聚集操作只需要调用 vgatherps 一次，而不考虑它访问的缓存行的数量。

```
#define MM_PACKSTORE4_PS(a, m, v)
_mm512_mask_extpackstorelo_ps(a, m, v,
_MM_DOWNCONV_PS_NONE, _MM_HINT_NONE)

__m512 p = zeroes;

__m512 xz01 = _mm512_mask_swizzle_ps(x, mask_0xCCCC, z,
_MM_SWIZ_REG_BADC);
__m512 yw01 = _mm512_mask_swizzle_ps(y, mask_0xCCCC, p,
_MM_SWIZ_REG_BADC);
__m512 xz23 = _mm512_mask_swizzle_ps(z, mask_0x3333, x,
_MM_SWIZ_REG_BADC);
__m512 yw23 = _mm512_mask_swizzle_ps(p, mask_0x3333, y,
_MM_SWIZ_REG_BADC);

__m512 tmp0 = _mm512_mask_swizzle_ps(xz01, mask_0xAAAA,
yw01, _MM_SWIZ_REG_CDAB);
__m512 tmp1 = _mm512_mask_swizzle_ps(yw01, mask_0x5555,
xz01, _MM_SWIZ_REG_CDAB);
__m512 tmp2 = _mm512_mask_swizzle_ps(xz23, mask_0xAAAA,
yw23, _MM_SWIZ_REG_CDAB);
__m512 tmp3 = _mm512_mask_swizzle_ps(yw23, mask_0x5555,
xz23, _MM_SWIZ_REG_CDAB);

_MM_PACKSTORE4_PS(&data[indices[0]], mask_000F, tmp0);
_MM_PACKSTORE4_PS(&data[indices[4]], mask_00F0, tmp0);
_MM_PACKSTORE4_PS(&data[indices[8]], mask_0F00, tmp0);
_MM_PACKSTORE4_PS(&data[indices[12]], mask_F000, tmp0);

// repeat stores for tmp1 with indices[1,5,9,13]
// repeat stores for tmp2 with indices[2,6,10,14]
// repeat stores for tmp3 with indices[3,7,11,15]
```

图 8-9 单精度数据的 SoA-to-AoS 转换的节选代码，使用的是 IMCI 内置函数

表 8-3 默认聚集 / 分散序列的指令数

	标量	128 位	256 位	512 位
# Elements	1	4	8	16
Load Indices	1	4	8	16
Gather	1	12	27	144

表 8-4 优化后的聚集 / 分散序列的指令数

	标量	128 位	256 位	512 位
Elements	1	4	8	16
Load Indices	1	4	8	16
Load Data	1	4	8	16
Shuffle	0	7	7	7

优化后的聚集 / 分散程序需要的指令数随着聚集元素的数量增长而增加。这并不是出乎意料的。重要的是原始序列和优化后序列在开销上的相对偏差，以及这可能对性能产生的影响，如图 8-10 所示。

处理器优化后的加速比为 1.23，协处理器优化后的加速比为 1.25。这些加速比都小于指令数量减少所带来的加速比，如表 8-3 和表 8-4 所示，但是由于现代架构的超标量和流水线特质我们不应指望数字是精确匹配的。此外，考虑到协处理器的特殊性，编译器产生的聚集操作

所需要的指令的数量不太可能同最坏情况下的数量一样高（由于之前的优化排序），而且由于通用和屏蔽寄存器的数量是有限的，可能引入我们不考虑的大量额外 mov 指令。

8.3.4 分摊聚集 / 分散和转换的开销

在 AoS 和 SoA 布局间进行即时转换可以显著减少聚集 / 分散操作的指令开销，但是快速的转换过程成为 kernel 的瓶颈，这个过程只对聚集的数据执行了很少的计算。在 miniMD 上构建邻居列表 kernel 是这项功能的一个好例子：对于每一个可能邻近的原子，只需要很少的算术运算和比较来判定这个原子是否应该加入到邻居列表中。

然而，不像计算力的 kernel（这里两个原子有相同邻居列表的可能性很小），邻居列表的构建把一些原子（在同一个 bin 中的）同一组固定的潜在邻居比较。因此，在多次重用 SoA 数据前，有可能在这些固定的潜在邻居组中只执行一次聚集和转换操作，如图 8-11 所示。

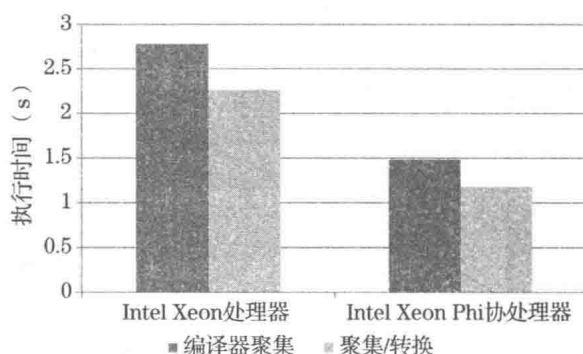


图 8-10 比较使用默认编译器的聚集操作计算短程力的执行时间和聚集 / 转置内部例程计算短程力的执行时间

```
int cached_bin = -1;
int ncache = 0;
for (int i = 0; i < nlocal; i++) {
    int* neighptr = &neighbors[i * maxneighs];
    int n = 0;
    const float xtmp = x[i*PAD+0];
    const float ytmp = x[i*PAD+1];
    const float ztmp = x[i*PAD+2];
    const int ibin = coord2bin(xtmp, ytmp, ztmp);

    if (ibin != cached_bin) {
        ncache = 0;
        for (int k = 0; k < nstencil; k++) {
            int jbin = ibin + stencil[k];
            int* loc_bin = &bins[jbin * atoms_per_bin];
            for (int m = 0; m < bincount[jbin]; m++) {
                const int j = loc_bin[m];
                scache[0*CACHE_SIZE+ncache] = j;
                scache[1*CACHE_SIZE+ncache] = x[j*PAD+0];
                scache[2*CACHE_SIZE+ncache] = x[j*PAD+1];
                scache[3*CACHE_SIZE+ncache] = x[j*PAD+2];
                ncache++;
            }
        }
        cached_bin = ibin;
    }

    for (int c = 0; c < ncache; c++) {
        const int j = scache[0*CACHE_SIZE+c];
        if (halfneigh && !ghost_newton && (j < i))
            continue;
        const float delx = xtmp - scache[1*CACHE_SIZE+c];
        const float dely = ytmp - scache[2*CACHE_SIZE+c];
        const float delz = ztmp - scache[3*CACHE_SIZE+c];
        const float rsq = delx*delx + dely*dely + delz*delz;
        if ((rsq <= cutneighsq)) neighptr[n++] = j;
    }

    numneigh[i] = n;
}
```

图 8-11 在 miniMD 上优化后的构建邻居列表的循环。为了提高可读性，我们此处展示的是标量代码（不是内置函数）

我们对于一个给定的箱使用事先分配的固定尺寸的数组 (scache) 来缓存转换模板。为了保证原子间的循环完整, 按照使用原子分解所需要的, 我们基于箱索引的变化检测是否执行聚集和转换操作 (图中用粗体强调) —— 粒子通过箱索引排序保证了模板的重用。第二个循环用和之前一样的方法把原子添加到邻居列表中, 但是直接从 scache 加载 SoA 数据而不是从位置数组 (x) 直接加载 AoS 数据。

除了重用转换邻居数据的能力, 这种构建邻居列表的方法有很多其他的好处。第一, 它提高了缓存的表现——而不是分散在内存中, 潜在的一组邻居连续地存储一个位置, 而且可能小到足够存储在接近计算单元 (即: L1 或者 L2) 大小的一级缓存里。第二, 它使得代码更容易向量化——模板索引的嵌套循环, 并且原始代码里箱的内容是整合进优化代码里 scache 内容的单次循环中, 这简化了完全填充向量寄存器的过程。这项优化方法对性能的影响如图 8-12 所示。

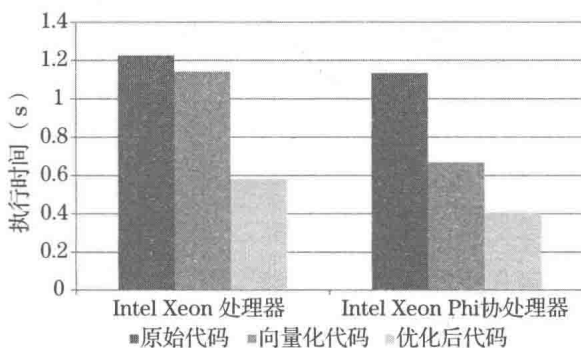


图 8-12 比较如下三种情况的执行时间: 使用原始标量代码构建邻居列表的执行时间, 即时转换的向量化实现的执行时间, 缓存转换结果的最佳实现方案的执行时间

8.4 总结

本章描述了聚集和分散操作并且解释了它们需要向量化特定循环的原因。由于它们相对于 SIMD 的连续加载和存储的高开销, 并且这是它们所不能避免的, 所以我们论证了 4 种优化技术来提高聚集 / 分散操作的性能: 使用域知识和排序来提高时间和空间的局部性; 基于它们的访问模式为热门计算循环选择一种适当的数据布局; 通过 AoS 和 SoA 之间的数据布局的即时转换, 来尽可能地减少每一个独立的聚集 / 分散操作的指令数; 利用多重循环迭代 (可能的情况下) 来分摊聚集 / 分散和转换操作的开销。

对于一个典型的分子动力学应用, 我们可以看到在最新一代的处理器和协处理器上未优化和优化后的代码版本之间的性能差距接近 2 倍。采用的两个优化方法在两个处理器平台上不但是一样的, 而且是基本通用的, 同时不局限于最新一代的 Intel 微处理器。我们期望它们能应用于未来的产品和其他的处理器设计, 因此在当今应用程序中优化聚集 / 分散模式能保证在将来它们依旧能表现出良好的性能水准。

8.5 更多信息

这里有一些我们推荐的同本章相关的附加阅读材料:

- 为分子动力学开发 SIMD, 使用 Intel Xeon 处理器和 Intel Xeon Phi 协处理器, <http://dl.acm.org/citation.cfm?id=2511458>。

- 分子动力学短程力的快速并行算法, <http://dl.acm.org/citation.cfm?id=201628>。
- Intel 64 和 IA-32 架构软件开发者手册, 第 2 卷: 指令集参考, A-Z, <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>。
- Intel Xeon Phi 协处理器指令集架构参考手册, <https://software.intel.com/sites/default/files/managed/68/8b/319433-019.pdf>。
- Intel 架构指令集扩展编程参考手册, <http://software.intel.com/sites/default/files/managed/68/8b/319433-019.pdf>。
- 下载本章和其他章的代码地址 <http://lotsofcores.com>。
- 下载最新版本的 miniMD, <http://mantevo.org>。

N 体问题直接法的众核实现

Alejandro Duran^{*}, Larry Meadows[†]

^{*} 西班牙, Intel 公司; [†] 美国, Intel 公司

9.1 N 体模拟

N 体模拟是一个常用的天体物理问题, 它需要计算交互粒子间重力影响下的空间移动。模拟中的每一个时间步中, 每个粒子实施于其他粒子的力根据下式计算:

$$F_{ij} = K \frac{m_i m_j (\vec{r}_i - \vec{r}_j)}{|\vec{r}_i - \vec{r}_j|^3}$$

基于此式, 每个粒子的速度和位置都将被更新。这一过程是迭代进行的, 并且在完成设定的时间步后结束。

虽然直接法 N 体内核因为其 $O(n^2)$ 的开销而并不在实践中采用, 但是它仍然用于在一个具有更低计算复杂度的复杂 N 体算法中计算一个相同子域中的粒子交互, 这些算法包括具有 $O(n \log n)$ 复杂度的 Barnes-Hut 算法或者其他方法的数值近似 (例如星群演化)。

9.2 初始解决方案

Vladimirov 和 Karpusenko 在 Intel Xeon Phi 协处理器上开发了一个简单的直接 N 体内核并进行了一些优化。我们扩展他们的工作, 在内核中引入软化因子 (即一个常用来调整邻近粒子间交互的常量), 并考虑具有不同质量的例子。

我们也研究了在单精度 (SP) 和双精度 (DP) 计算下不同元素数量配置下的性能区别。

图 9-1 中展示了初始解决方案, 它包括两个由 Vladimirov 和 Karpusenko 提出的优化。

- 把代码从数组结构转变为结构数组。由于同一个域的元素位于内存中的连续位置, 因此这个改进允许更高效的向量化实现,
- 通过 `-fp-model fast=2` 编译选项来使编译器生成不支持 IEEE 精度的代码。该选项允许编译器生成更短的代码序列, 但是这会导致精度降低。为了在双精度中改进性能, 我们显式要求使用单精度 `sqrtr` 版本。对双精度版本, 即使调用 `-fp-model fast=2` 编译选项, 编译器也会生成更长的代码。

实践中, 我们发现将 OpenMP 循环调度设置为动态类型会达到最高性能。这可能是由于 L2 缓存缺失导致的内存访问带来的负载不均衡。也需要注意, 对于表达式 `1.0f/sqrtrf`, 编译器自动使用 Intel Xeon Phi 协处理器提供的平方根倒数原语。

图 9-2 展示了 N 体模拟的时间步长, 每个时间步长都会调用上述内核函数, 并且不同粒子的位置将会更新。


```

template <class T>
void computeForces(ParticleSystem<T> p, const T dt)
{
    int n = p.nParticles;
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < n; i++) {
        T Fx = static_cast<T>(0.0);
        T Fy = static_cast<T>(0.0);
        T Fz = static_cast<T>(0.0);
        for (int j = 0; j < n; j++) {
            const T dx = p.x[j] - p.x[i];
            const T dy = p.y[j] - p.y[i];
            const T dz = p.z[j] - p.z[i];

            const T drSquared = dx*dx + dy*dy + dz*dz
                               + p.softening;
            const T drPowerN12 = 1.0f / sqrtf(drSquared);
            const T drPowerN32 = drPowerN12 * drPowerN12 *
                               drPowerN12;
            const T s = p.m[j] * drPowerN32;

            Fx += dx * s; Fy += dy * s; Fz += dz * s;
        }
        p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;
    }
}

```

图 9-1 初始的 N 体内核

```

for (int iter = 0; iter < nIters; iter++) {
    computeForces(p,dt);
    for (int i = 0; i < p.nParticles; i++) {
        p.x[i] += p.vx[i] * dt;
        p.y[i] += p.vy[i] * dt;
        p.z[i] += p.vz[i] * dt;
    }
}

```

图 9-2 时间循环步长

本章中的实验平台是 Intel Xeon Phi 协处理器 7120P (除非说明, 否则 turbo 功能始终关闭)。它包含时钟频率为 1.23GHz 的 61 个内核, 总共支持 244 个线程, 单精度最高峰值性能为 2.4TFLOPS, 双精度为 1.2TFLOPS。协处理器包含 16GB GDDR5 内存。每个核配备 32KB L1 缓存和 512 KB L2 缓存。

编译器为 Intel C/C++ 14.0.2 版本编译器, 编译选项为 `-openmp -mmic -fp-model fast=2 -O3`。测量性能前, 执行一次未计时的迭代以便预热缓存。所有性能数值的单位都是每秒处理的交互数, 以 G 为单位 (亦即每秒中有多少对粒子被更新, 以十亿计数)。为了计算 GFLOPS 数, 这一数值可以乘以 20, 这是由于每对交互的计算包含 20 个浮点操作。

图 9-3 展示了初始版本的单精度和双精度性能。在单精度中, 虽然在 10000 ~ 60000 个粒子时的性能比较合理, 但是更多的粒子会导致性能下降。对于双精度, 也可以在粒子数超过 40000 后观察到同样的趋势, 只是这种下降趋势更为缓慢。

9.3 理论极限

优化应用的一个常见的问题是需要理解目标 (亦即需要理解何时需要停止进一步优化?)。我们可以构建应用的一个理论模型来确定需要执行多少次操作, 以及可能达到的最大性能 (对一个给定的应用程序也称为“光速”)。

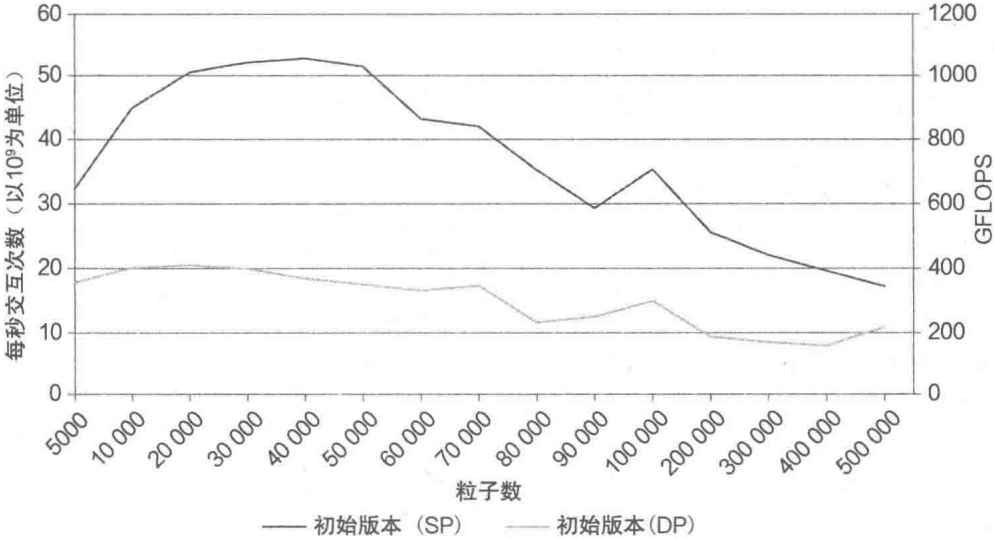


图 9-3 初始版本的单精度和双精度性能

我们现在应用这种方法到 N 体问题。 N 体问题内核的主要计算量在于粒子间力的计算，复杂度为 $O(n^2)$ 。更新所有粒子的位置的复杂度为 $O(n)$ 。力交互的计算主要是位于内层的 j 循环（在每个 i 循环的末尾进行更新的复杂度为 $O(n)$ ）。如果考虑图 9-1 中的代码，在 Intel Xeon Phi 协处理器上将其映射到 Knights Corner 指令集 (KNCi) 将得到：

- 三次加法 (vsub)
- 三次乘加 (vmadd)
- 一次逆平方根 (vrsqrt)
- 三次乘法 (vmul)
- 三次乘加 (vmadd)

这会产生图 9-4 所示的单精度代码，亦即上述代码序列的直接翻译，仅增加了软化数值的广播操作。由于 vmadd 指令的破坏性，数值需要在每次迭代后修复。

```
vsubrps    (%r14,%r10,4), %zmm7, %zmm12
vbroadcastss .L 2il0floatpacket.2(%rip), %zmm8
vsubrps    (%r9,%r10,4), %zmm6, %zmm13
vmadd231ps %zmm12, %zmm12, %zmm8
vsubrps    (%rbx,%r10,4), %zmm5, %zmm15
vmadd231ps %zmm13, %zmm13, %zmm8
vmadd231ps %zmm15, %zmm15, %zmm8
vrsqrt23ps %zmm8, %zmm10
vmulps     (%r15,%r10,4), %zmm10, %zmm9
vmulps     %zmm9, %zmm10, %zmm11
vmulps     %zmm11, %zmm10, %zmm14
vmadd231ps %zmm12, %zmm14, %zmm2
vmadd231ps %zmm13, %zmm14, %zmm1
vmadd231ps %zmm15, %zmm14, %zmm3
```

图 9-4 没有预取和循环计数的计算交互力的内层 j 循环的单精度代码

假设所有访存操作都在 L1 缓存中命中，每个指令的延迟是 4 个周期，因此一个线程的计算总共花费 52 个周期来计算 16 个粒子的交互（16 是协处理器中单精度向量寄存器的宽度）。幸运的是，处理内核上其他三个线程可以完美地填满空闲的计算周期。因此对于每个内核的 4 个线程，它用 56 个周期来计算 64 个粒子的交互（除了最后一次迭代，它还需要 3

个周期来完成所有线程的计算。对于大量粒子的情况，我们可以忽略这一开销)。因此，每个周期每个内核计算 1.142 个单精度粒子交互。

对双精度而言，我们可以在打包的双精向量上使用同样的代码，除了计算平方根倒数操作之外，向量宽度均为 8。第一代 Intel Xeon Phi 协处理器只支持单精度 `vrsqrt`，因此双精度向量需要转变为单精度向量。如果我们循环展开循环为 2，可以打包 16 个元素，即从两个双精度向量变为一个单精度向量并调用 `vrsqrt`。图 9-5 展示了计算平方根倒数以及双精度变换的代码，这需要 36 个周期。使用循环展开，需要额外的 104 个周期，因此每个使用 4 线程的处理器内核一共需要 140 个周期来计算 64 个元素。双精度情况下每个周期每个内核平均处理 0.457 个粒子交互。

```
vcvtpd2ps {rn}, %zmm5, %zmm14
vcvtpd2ps {rn}, %zmm3, %zmm4
vblendmps %zmm4, %zmm6, %zmm15{%k3}
vpermf32x4 $68, %zmm14, %zmm15{%k1}
vrsqrt23ps %zmm15, %zmm16
nop
vpermf32x4 $238, %zmm16, %zmm17
vcvtps2pd %zmm16, %zmm20
vcvtps2pd %zmm17, %zmm22
```

图 9-5 在双精度（具有受限的精度）中使用 `vrsqrt`

下一代 Intel Xeon Phi 协处理器 Knights Landing 将会支持 AVX-512 指令集，它包含双精度平方根倒数指令。这将会使双精度中的交互次数提高到每个周期每个内核处理 0.571 次交互，并且消除双精度转换为单精度时的损失。

结合这个成本模型以及内核数量和每个不同 Intel Xeon Phi 协处理器的时钟频率，我们可以得到每秒可以处理的最大交互次数，结果如图 9-6 所示。

模型	内核	频率(GHz)	最大值每秒交互 次数 (以G为单位) (单精度)	最大值每秒交互 次数 (以G为单位) (双精度)
7120 ×	61	1.238	85.7	34.3
7120 × (具有 turbo)	61	1.333	92.9	37.2
5110 ×	60	1.053	72.2	28.9
3120 ×	57	1.1	71.7	28.7

图 9-6 不同 Intel Xeon Phi 协处理器上每秒处理的交互次数上界（以 10^9 为单位）

从图 9-6 中所示的最优上界可知，最初版本的性能仍然有较大的提升空间。单精度性能是每秒 52.7×10^9 次交互，上界是 85.7（61%）。双精度性能是每秒 20.5×10^9 次交互，上界是 34.3（59%）。

9.4 降低开销和对齐数据

下一个优化解决了检查初始代码后发现的两个问题。

- OpenMP 的 `parallel` 构件位于 `computeForces` 例程的内部。这有两点影响。首先，更新位置向量的操作将会被序列化，由于 `computeForces` 的复杂度占支配地位，所以这点并无大碍，但是当粒子数目较少时会导致性能显著降低。其次，每一个时间步长都会产生并行区域。虽然 Intel OpenMP 运行时维持底层线程的运行并且可以重用，但是这仍然会在粒子数量较少时导致不可忽视的开销。

- 数据不是对齐的，也不会指示编译器发布对齐指令。对于编译器而言，数据对齐是保证其产生对齐的向量内存操作读写和在向量操作上使用内存操作数的关键。没有数据对齐，我们无法使用前一节中提到的最优的指令序列。

图 9-7 展示了这一版本中的初始化和时间步长代码的变化。数据通过调用 `_mm_malloc` 来保证对齐。OpenMP 的 `parallel` 构件放置在外层循环，并且在更新粒子位置的代码中放置 OpenMP 的 `for` 构件。注意，每个 `for` 循环尾部的隐式栅栏会导致线程间同步的迭代（每个线程似乎有相同的 `iter` 数值并且迭代相同的次数）。

```
... = (T *) _mm_malloc(bytes, 64);
...
#pragma omp parallel
for (int iter = 0; iter < nIters; iter++) {
    computeForces(p, dt);
    #pragma omp for
    for (int i = 0; i < p.nParticles; i++) {
        p.x[i] += p.vx[i] * dt;
        p.y[i] += p.vy[i] * dt;
        p.z[i] += p.vz[i] * dt;
    }
}
```

图 9-7 第二个版本的初始化和时间步长循环

图 9-8 展示了 `computeForces` 内核的变化，代码使用 OpenMP 的 `for` 构件而不是 `parallel` 构件。`vector aligned` 语句用来指示编译器只发出对齐数据的代码，这意味着数据必须对齐，否则程序将会出现硬件异常。

```
template <class T>
void computeForces(ParticleSystem<T> p, const T dt)
{
    int n = p.nParticles;
    #pragma omp for schedule(dynamic)
    for (int i = 0; i < n; i++) {
        ...
        #pragma vector aligned
        for (int j = 0; j < n; j++) {
            ...
        }
        ...
    }
}
```

图 9-8 第二个版本的 N 体内核代码

图 9-9 和图 9-10 分别展示了单精度和双精度性能。点线表示前一节计算得到的性能上界。可以看到，当粒子数目较少时，特别是针对单精度而言，`parallel` 构件移动到外层循环使性能提高。使用对齐指令也极大地增加了单精度（最高到 67.5）和双精度（最高到 26.9）的性能。

即使应用这些优化，性能依然会在大致相同的粒子数时降低，这又是什么因素导致的呢？

利用 Intel VTune Amplifier 分析程序可知，L2 间存在极大的数据传输。图 9-11 展示了 L2 间传输数的占比，左侧 y 轴是所有请求的百分比数量，并且也展示了每个内核的四个内层 `j` 循环（右侧 y 轴）数组的 L2 的空间需求。图 9-11 还展示了即使对于粒子数目为 40000 这么小的数量，L2 缓存也已经用尽。由于需求的数据通常来自于其他内核的 L2 缓存，因此也导致 L2 间传输的增加。从其他内核的 L2 缓存读取数据要快于从内存中读取数据。但是数据传输的次数随着粒子数的增加而增加，并且会导致程序性能下降。

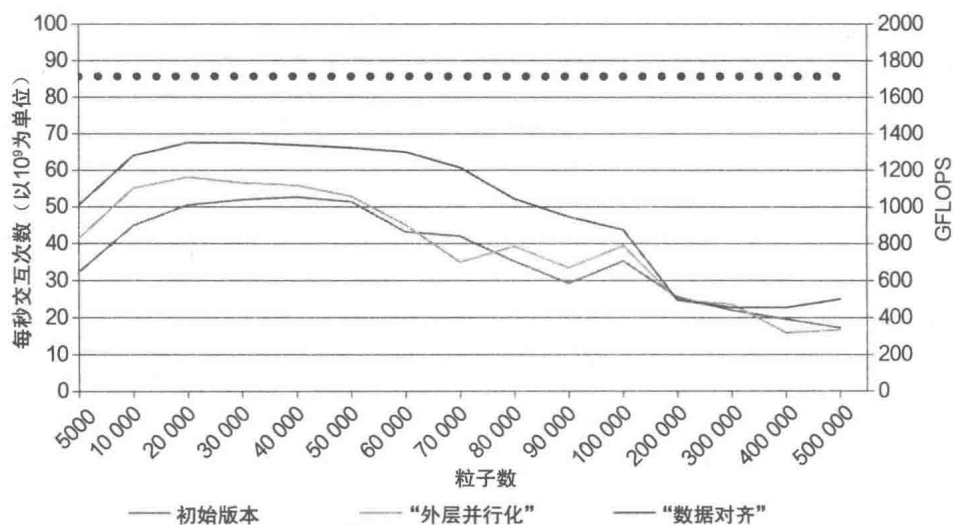


图 9-9 第二个版本代码的单精度性能

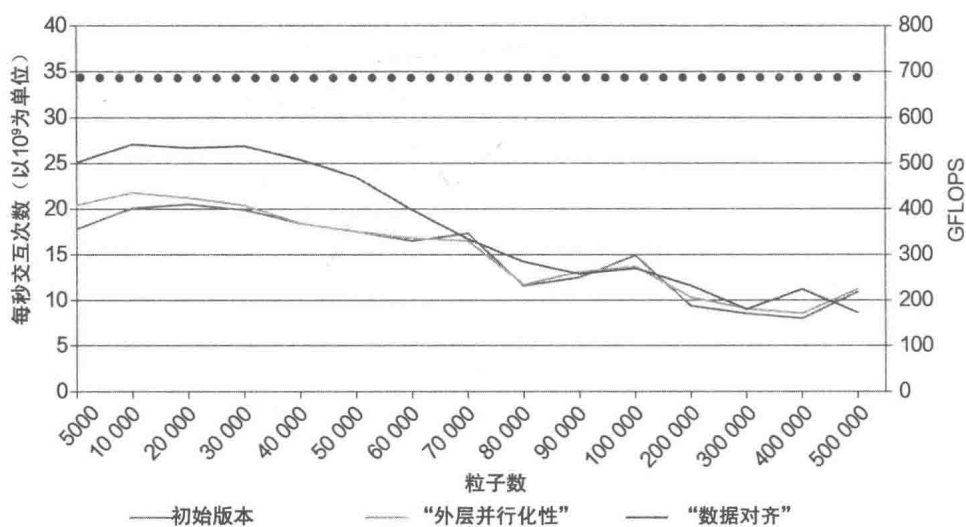


图 9-10 第二个版本的双精度性能

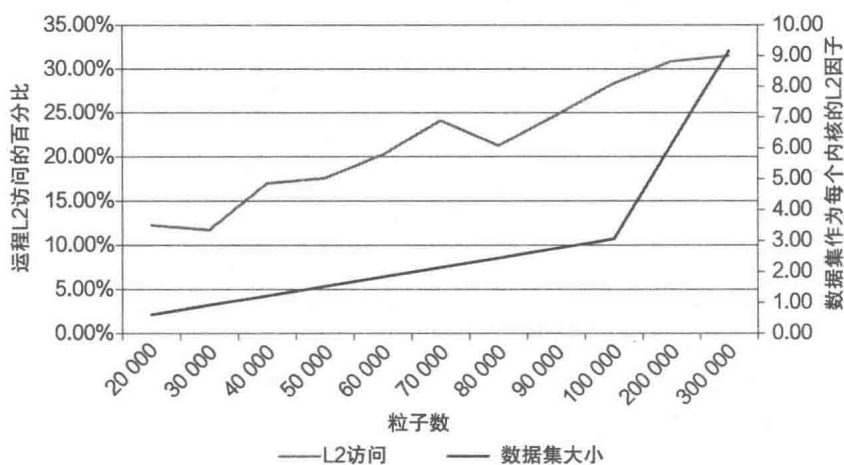


图 9-11 第二个版本的 L2 访存压力

9.5 优化存储层次

我们利用熟知的循环分块技术来提高局部性以及更高效地在每个 i 循环迭代中重用粒子。特别地，我们在 i 循环和 j 循环使用正方形分块，粒子数大小为 BF。每个分块都分配给一个线程，如果分块大小合适， j 分块的元素将会在 i 分块所有的元素上重用，避免了被替换出缓存。这将提高数据局部性。

图 9-12 展示了这一内核代码的优化，以及分块因子的计算。OpenMP 的 collapse 语句用于 for 语句中以便将所有分块分配给线程。由于多线程可能同时更新速度向量，因此需要利用 OpenMP 原子构件。由于原子操作位于 i 层循环，因此可以预计其将不会导致更大的性能下降。

```
int align = CACHELINE_SIZE/sizeof(T);
BF = nParticles / sqrt(omp_get_max_threads()) / 4;
BF = (BF+align-1)/align * align;
if ( BF > max_block<T>::value ) BF = max_block<T>::value;
...
int n = p.nParticles;
#pragma omp for collapse(2)
for ( int ii = 0; ii < n ; ii += BF )
for ( int jj = 0; jj < n; jj += BF ) {
    int imax = min(ii+BF,n);
    int jmax = min(jj+BF,n);
    for (int i = ii; i < imax ; i++ ) {
        ...
        #pragma vector aligned
        for (int j = jj; j < jmax; j++) {
            ..
        }
        #pragma omp atomic
        p.vx[i] += dt*Fx;
        #pragma omp atomic
        p.vy[i] += dt*Fy;
        #pragma omp atomic
        p.vz[i] += dt*Fz;
    }
}
```

图 9-12 二维分块 N 体内核代码

注意，这里删除了动态调度语句，这是由于使用默认的静态调度能获得更好的性能。我们认为这是由于改进的内存行为已经避免了前面由于线程访问核间互联的负载不均衡，于是不再需要动态调度来均衡线程之间的工作负载。

选择合适的分块大小 BF 是一项需要技巧的问题。在每个 j 迭代中，我们需要访问三个分块： i 分块、当前的 j 分块以及下一次迭代需要预取的 j 分块。处理器内核上 4 个线程中的每一个都利用一个包含 4 个向量 (x 、 y 、 z 和 m) 的分块。一种简单的计算方法是：

$$BF = \frac{L2 \text{ 大小}}{\text{向量数} * \text{每个内核的线程数} * \text{类型的大小} * \text{分块数}}$$

因此，512KB 大小的 L2 缓存意味着最大的单精度分块大小为 2730 个元素，双精度时元素个数减半。

另一个影响分块大小的因素是 BF 必须是数据对齐类型的整数倍，否则每个分块中的一些元素需要在分块主要向量化部分的前面和后面进行单独处理。因此分块大小不是对齐数据类型的整数倍时会导致极大的开销。同时考虑这些因素，我们需要选择最大的单精度分块大小为 2688，双精度时元素个数减半。

压缩和并行化分块循环也意味着总共可以分配给线程的可用迭代次数减少分块大小 $1/BF$ ，在粒子数目较少时会导致数目低于可用线程总数。例如 30000 个粒子使用最大分块大小时，可分配给线程的迭代数目为 144，它小于 Intel Xeon Phi 协处理器上的可用线程数目。

为了避免这种情况，需要计算能保证每个可用线程至少处理一个或多个迭代的分块大小，这可以利用下式得到

$$BF = \min \left(\max BF, \frac{n\text{Particles}}{\sqrt{\text{线程数}} * \text{松弛因子}} \right)$$

公式中的松弛因子 (slack factor) 用来保证每个线程至少有一次或多次迭代，降低了负载不均衡可能的影响，这里设置松弛因子为 4。

另一个需要进一步考虑的优化是利用 L2 缓存在同一个处理器内核的所有线程间共享这一特性，我们可以将同一个分块分配给处理器内核中的所有线程（而不是为每个线程分配一个分块）。共享分块意味着 L2 缓存空间可以支持更大的分块，这一方式可以使需要填满整个协处理器的并行任务数量降低 $1/4$ 。考虑所有这些因素，单精度情况下使用的最大 BF 值为 4096，双精度时元素个数减半。

但是，OpenMP 对没有使用嵌套并行的共享分块方式仅提供有限的支持。当前 OpenMP 实现版本中嵌套并行的开销较大，因此，我们设计了如图 9-13 所示的一个版本，即循环压缩和工作分配通过手工方式实现。核心在于外层循环的分配使用内核标识符完成，因此同一处理器内核上的所有线程将会得到同样的迭代任务，并且内层 i 循环线程通过内核标识符和间隔大小 4 来处理不同的迭代。注意，这个版本的优化代码假定 OpenMP 线程连续地位于协处理器硬件线程上，这可以利用 KMP_AFFINITY 环境变量实现。

```
core = omp_get_thread_num() / core;
// compute thread number inside core
cid = omp_get_thread_num() % 4;
// compute core_start and core_end as in schedule(static)
...
for ( l = core_start; l < core_end ; l++ ) {
    int ii = (l/nb) * BF;
    int jj = (l%nb) * BF;

    int imax = min(ii+BF,n);
    int jmax = min(jj+BF,n);
    for ( int i = ii+cid; i < imax ; i+= 4 ) {
        ...
        #pragma vector aligned
        for (int j = jj; j < jmax; j++) {
            ...
        }
    }
}
```

图 9-13 每个分块使用多线程的 N 体问题代码

图 9-14 和图 9-15 分别展示了单精度和双精度下的两种分块方法（具有内核内并行性以及不具备该并行性）的性能结果。注意，两种版本的性能都随着粒子数目的增加而增加。但是只有在粒子数目非常大时其性能才超越前一版本。为了实现最优的应用程序性能，两个版本需要同时支持粒子数目很小以及非常大的两种情况。另外注意，利用内核内并行性的单精度版本性能更好，但是双精度版本的性能更稳定。

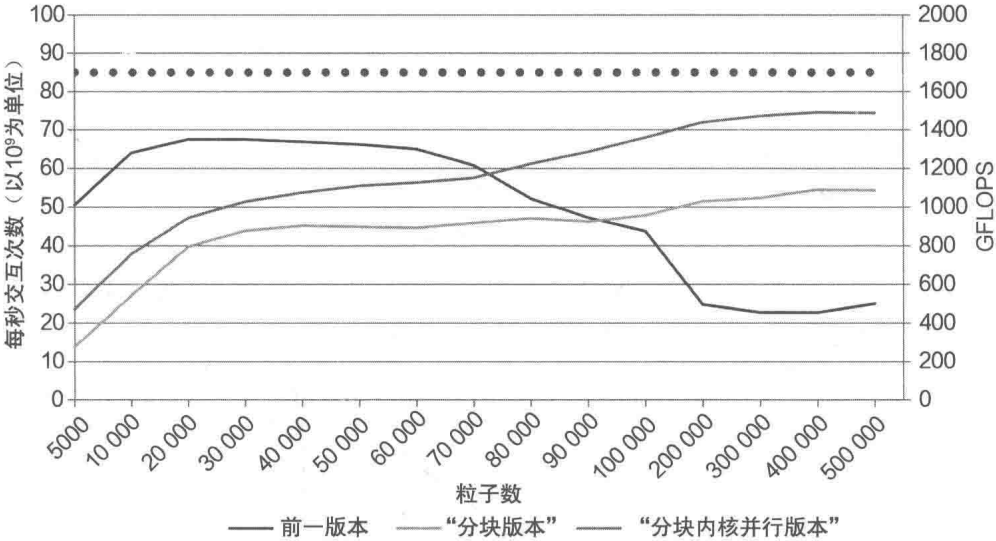


图 9-14 单精度分块版本的性能

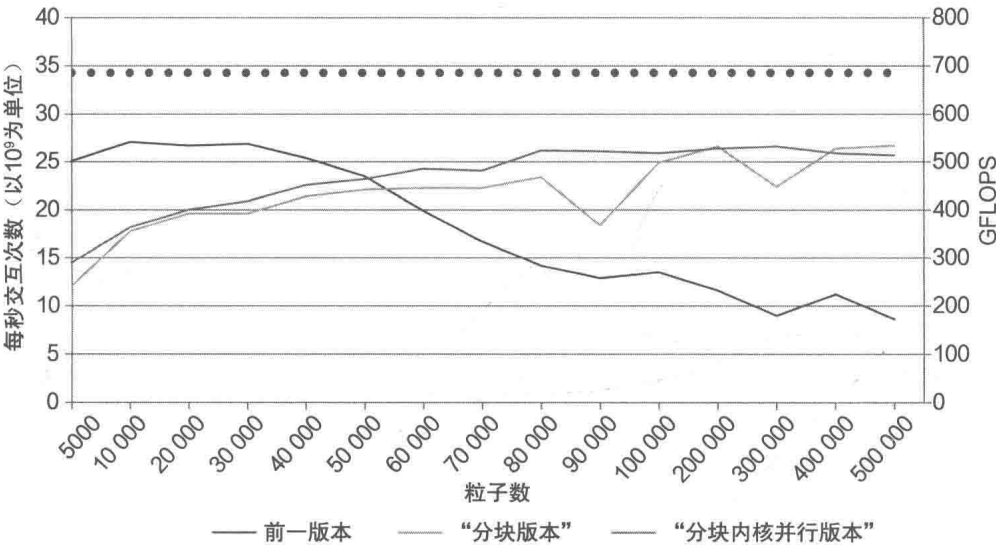


图 9-15 双精度分块版本的性能

9.6 改进分块

另一种方法是使用两个不同的分块因子：BFI 是 i 层循环的分块因子，BFJ 是 j 层循环的分块因子。使用两种分块因子意味着 i 层循环可以选择提供足够多的并行性以避免使用 OpenMP 的 collapse 语句，同时也提供选择 j 循环分块大小的随意性，允许优化 L2 缓存的访问。

图 9-16 展示了实现这种方案的 N 体问题内核代码。由于只有一个线程处理每

```
#pragma omp for
for ( int ii = 0; ii < n ; ii += BFI ) {
    imax = min(n,ii+BFI);
    for ( int jj = 0; jj < n; jj += BFJ ) {
        jmax = min(n,jj+BFJ);
        for ( int i = ii; i < imax; i++ ) {
            ...
            #pragma vector aligned
            for ( int j = jj; j < jmax; j++ ) {
                ...
            }
        }
    }
}
```

图 9-16 具有两个分块因子的 N 体内核函数

个 i 迭代, 因此这里可以删除原子语句。

图 9-17 展示了 BFI 和 BFJ 分块因子的计算过程。BFI 根据所有线程在 ii 循环上都只拥有一次 i 迭代进行计算, 并且保证缓存行对齐以便改进性能。缓存行对齐的一个不利影响是有些线程可能没有足够的迭代计算, 因此缓存对齐的分块大小及其导致的负载不均衡间需要进行权衡。通过观察可知, 当对齐的缓存大小导致最多 2% 的负载不均衡时性能最优。否则, 需要选择保证最大负载均衡的非对齐分块大小。

```
int nthreads = omp_get_max_threads();
BFI_unaligned = (nParticles+nthreads-1)/nthreads;

int align = CACHELINE_SIZE/sizeof(T);
BFI = (BFI_unaligned+align-1)/align*align;

int iters = nParticles / BFI;
if ( nParticles % BFI != 0 ) iters++;
float balance = ((float) iters) / nthreads;
if ( balance < 0.98 ) BFI = BFI_unaligned;

BFJ = bf<T>::max_value;
float
l2_occupancy=((float)nParticles)*sizeof(T)*4)/(512*1024);
if ( l2_occupancy < 4 ) BFJ *= 2;
if ( l2_occupancy < 2 ) BFJ *= 2;
```

图 9-17 N 体内核函数改进了两个分块因子的计算

对单精度而言, BFI 选为 4096, 对双精度而言是 2048。下面建议一个附加的优化: 在不同的时间点, 同一个处理器内核上的线程在遍历 j 循环时, 可能会使用相同的数据, 这种内核内数据共享的概率随着模拟中粒子数目的降低而增加。特别地, 当工作集中总粒子数目不是 L2 大小的 4 倍时, 至少有两个线程共享粒子。当工作集大小小于 L2 大小的两倍时, 4 个线程都将共享粒子数据。这种情况下需要增加分块大小两倍或者四倍。

图 9-18 和图 9-19 分别展示了使用上一段中 BFI 和 BFJ 两个分块因子大小的性能。特别是, 在粒子数较小时模拟程序性能有较大的提升, 并且在粒子数目较大时性能也有一定的提升。只有在粒子数小于 5000 的情况下, 未分块版本的性能最优。

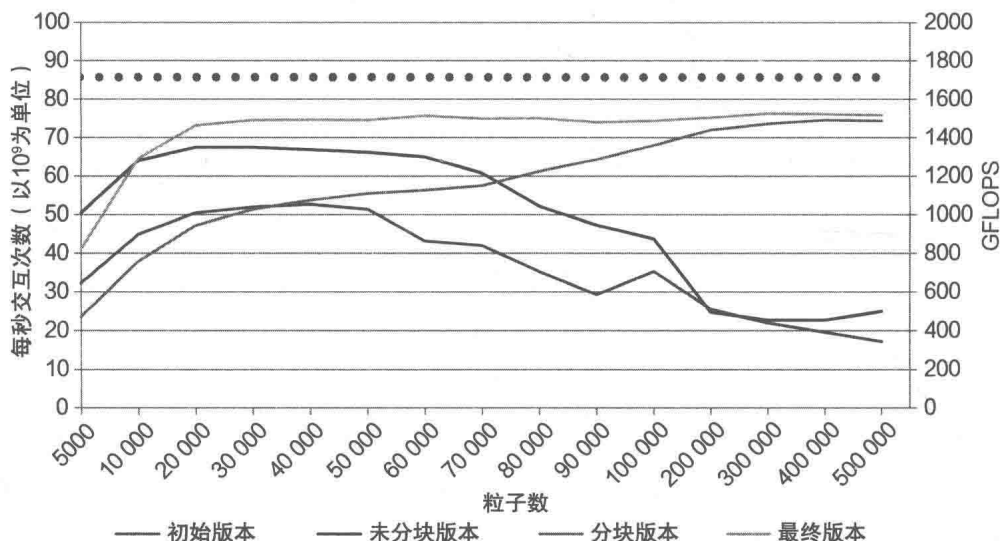


图 9-18 不同优化步的单精度性能

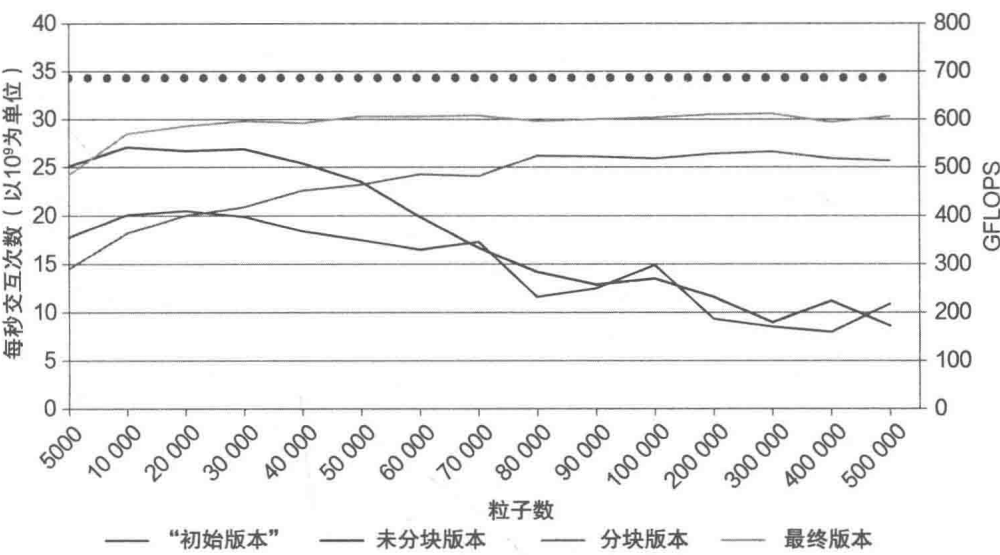


图 9-19 不同优化步的双精度性能

通过本章介绍的一系列优化过程，单精度和双精度的性能都达到了性能模型预测理论峰值的 89%。单精度性能达到了每秒处理 76.3×10^6 次交互数，双精度性能达到了每秒处理 30.6×10^6 次交互。

用绝对性能表示，单精度版本达到 1.53TFLOPS，即协处理器峰值性能的 63.72%；双精度版本达到 612GFLOPS，即协处理器峰值性能的 50.9%。

9.7 主机端的优化

图 9-1 和图 9-2 所展示的初始版本没有附加任何体系结构信息。你可能希望得知前面小节中的一系列优化如何影响了 Intel Xeon Phi 协处理器上最终 N 体内核函数的性能。

图 9-20 和图 9-21 展示了双插槽 E5-2697v2 Intel Xeon 处理器上 N 体问题初始版本以及图 9-16 和图 9-17 中展示的最终版的性能对比，该系统总共有 24 个处理器内核，每个内核上有两个线程。代码没有任何变化，只是编译选项将 -mmic 变为 -xHost。所有的 48 个线程都被用到以便获得最大性能。

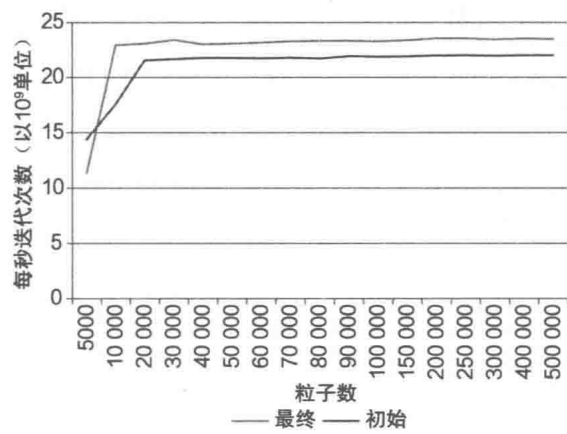


图 9-20 不同优化步的单精度性能

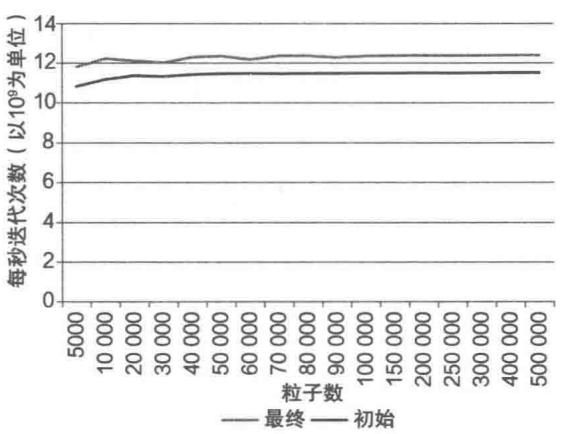


图 9-21 不同优化步的双精度性能

可以看到,在图 9-20 和图 9-21 中,对单精度和双精度大概都有 10% 的性能提升,这种提升来自数据对齐以及使用对齐的向量指令。如果查看最初版本的性能,对于大规模粒子数并没有像协处理器上程序那样的性能下降。这表明在 Xeon 处理器上运行时,提供处理器内核所需要的数据并不需要过多的优化(这可以部分地解释为,由于访存压力较小并且配置最后一层共享缓存会得到更好的性能),并且这样解释了为何循环分块没有影响性能(它提高了 L1 和 L2 缓存的命中率,但是并没有转变为性能提升)。其他优化也没有提升或者降低性能,因此我们有一个在主机端和协处理器端均高效运行的内核版本。

9.8 总结

本章展示了一个 Intel 众核处理架构上优化的 N 体内核代码。该优化版本在传统处理器上也获得了性能提升。在 Intel Xeon Phi 协处理器上的重要优化包括:

- 高效向量化,合适的数据对齐,使用向量超越函数和数据布局。
- 提供足够的并行性,降低开销,避免负载不均衡。
- 通过循环分块来有效利用存储层次。

这些优化使单精度最大性能为从每秒处理 52.7×10^6 次交互计算增加到 76.3×10^6 次(44% 的提升),双精度最大性能为每秒处理 20.5×10^6 次交互计算增加到 30.6×10^6 次(66% 的提升)。使用同样的代码,这些优化也在 Intel Xeon 处理器上获得了 10% 的性能提升。

本章也讨论了如何快速为优化过程确定一个目标,最终的优化版本达到了峰值上界的 89%,因此这是一个较为合理的终点。

我们还展示了在 Intel Xeon Phi 协处理器上的所有这些优化也能够在 Intel Xeon 处理器上获得性能提升。

9.9 更多信息

下面是有关本章讨论内容的附加阅读材料的列表:

- Vladimirov,A.,Karpusenko,V.,2013.Test-driving Intel® Xeon Phi™ coprocessors with a basicN-body simulation. <http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx>.
- N-Body problem, http://en.wikipedia.org/wiki/N-body_problem.
- Arora,N.,Shringarpure,A.,Vuduc,R.W.,2009. Direct N-body kernels for multicore platforms. In the International Conference on Parallel Processing ICPP'09.
- Rodionov,S.A.,Sotnikova,N.Ya.,2005. Optimal choice of the softening length and time step inN-body simulations. Astronomy Reports 49 (6),470-476.
- Loop tiling, http://en.wikipedia.org/wiki/Loop_tiling.
- 本章及其他章的代码下载地址 <http://lotsofcores.com>。

N 体方法

Rio Yokota, Mustafa AbdulJabbar

沙特阿拉伯，阿卜杜拉国王科技大学

本章讨论的优化技术包括使用编译器选项及指令，对代码进行最小的修改以提高性能。本章将展示无须重写代码，编译器即可产生融合的乘加以及平方根倒数操作，并展示双循环的外层循环仅通过简单的编译指令即可向量化。本章将比较使用编译指令产生的代码及使用内置函数（intrinsic）手工调优的代码，也将比较同一代码分别在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上的性能。无需重写原有 C 代码，本章将使程序在协处理器上达到 1.5TFLOPS 的单精度性能。本章将使用直接 N 体方法来证明上述优化的效果。 N 体方法传统上应用于基于粒子的模拟，比如天体物理学和分子模拟，近期扩展到求解更一般形式的偏微分方程。关于这些使用的参考内容可以查看 10.6 节。由于每加载 $4N$ 个浮点数要执行 $20N^2$ 操作，因此 N 体方法的内核具有高计算强度。比较 N 体方法与矩阵乘法可以发现， N 体方法的浮点操作 / 字节比率更高。这种高强度计算使 N 体方法在未来体系结构中仍将在计算上受限。本章将重点研究 N 体方法在 Intel Xeon Phi 协处理器上的优化，并展示这些优化在 Intel Xeon 处理器上的效果。

10.1 快速 N 体方法和直接 N 体内核

直接 N 体内核计算所有 N 个主体与 N 个主体成对的相互作用，因此运算量为 $O(N^2)$ 。据 Barnes 等人所述（见 10.6 节），快速近似法使用主体的层次化域分解法，以及内核的截断级数展开可以使操作量降到 $O(M \log N)$ 甚至 $O(N)$ 。这种 $O(N)$ 的方法通常称为快速多极方法（Fast Multipole method, FMM）。这种快速 N 体法背后的原理是，远场的主体不必单独考虑，而是分成多极。然而，近场的主体仍必须使用直接 N 体内核进行准确计算。因此，直接 N 体内核的性能对快速 N 体法是至关重要的。

计算复杂性指当问题规模增大时所执行计算操作的渐近量。计算强度指每次加载的数据量所需执行的计算操作量。结合 FMM 的 $O(N)$ 最优复杂度以及其中直接 N 体内核的高计算强度使 FMM 成为未来体系结构上很多算法有趣的替代品。这是因为类似稠密线性代数的常规高计算强度算法也易具有高计算复杂性。这些算法在计算上是受限的，可以利用大多数的运算单元，但开始阶段在计算上的花费是非常昂贵的。另一方面，那些具有低计算复杂性的方法，比如快速傅里叶变换（FFT）和稀疏线性代数，运算强度低。它们非常有效，但不能达到现代处理器峰值运算性能的较高百分比。FMM 非常好地结合了 $O(N)$ 的复杂性及比矩阵乘还要高的计算强度，这意味着 N 体方法提供了一个随问题规模顺利扩展的有效算法，但在接下来几十年的未来体系结构上仍将在计算上受限。

直接 N 体内核很容易从 FMM 分离，因为它通常在邻居查找例程中作为一个函数调用，这个函数调用占用了 FMM 大部分的执行时间。因此，如果能够优化直接 N 体内核，它就

可以自动优化 FMM 的热点。直接 N 体内核只有几十行代码，即使在汇编级优化工作量也不大。本章将通过一个非常具体的例子来展示只用编译器选项可以达到的性能，然后介绍如何使用内置函数从直接 N 体内核中取得更多性能。

10.2 N 体方法的应用

在物理现象本身可以描述为离散点集时， N 体方法可以自然地应用到这类问题，甚至早于数值离散化。引力或静电力下的多体问题是一个典型的例子，恒星和原子可分别表示为质量和静电荷的点源。 N 体方法可以通过离散化从离散场延伸到连续场，使得这些方法可用于解决结构力学、流体力学、电磁学、声学，甚至量子力学中的问题，但适用性并不意味着它是解决特定问题的最佳方法。

快速 N 体法适用于几何信息动态变化的应用中。如果几何信息是静止的，更明智的方法是将这一信息存储在矩阵形式中，并在该同一矩阵反复执行稠密或稀疏线性代数操作。 N 体法可以视为“无矩阵”法，在该方法中，矩阵将在与源点向量相乘之前动态形成。很明显，这种方法仅当矩阵 / 几何信息频繁变化时才发挥优势，因为此时存储这些信息不会节省任何计算。基于粒子的方法正是这种情况，因为每个时间步长粒子的位置都会前进。如果系统是非常动态的，自适应网格细化也可能导致相似的几何信息更新量。

有很多因素能够影响 N 体方法与其他椭圆偏微分方程 (Partial Differential Equation, PDE) 解法的比较优势，比如 FFT 和多网格法。渐近常量在决定这些不同 $O(N)$ 或 $O(N \log N)$ 算法的相对性能时起到关键作用。众所周知，FFT 最少可达到 $2N \log N$ 次操作，多网格法最少可达到 $5N$ 次操作。而典型 FMM 的渐近常量要大得多，但在指示点位置时使用平移或转动对称性能显著减少这一常量。

很难比较 FFT 和 FMM，因为 FFT 对每个未知数有更高的空间分辨率，因此 N 相同时的比较并不公平。但对于有局部特征或非连续的场，FFT 的同构空间分辨率无疑成了劣势。此外，当通过指示点位置将平移或转动对称性应用于 FMM 时，FMM 可以使用 BLAS 3 操作，比多网格法更加有效。因此在这种情况下，解决相同精度的相同问题时 FMM 比多网格法更快。

传统的 FMM 需要格林函数解，因此它能够处理的科学应用类型仅限于那些有格林函数的应用。将 FMM 泛化到矩阵的层次化低阶近似使相同的框架能够应用到更广泛的应用。这些方法使用例如显秩 (rank-revealing) QR，截断奇异值分解 (truncated SVD)，或自适应交叉近似的低阶近似方法，而不是多极扩展。这使 FMM 不再依赖于格林函数的存在性，从而可以应用在原始 FMM 无法解决的问题，例如变系数柏松方程或协方差矩阵。

在预测 FFT、FMM 及多网格在未来体系结构上的性能时，一个有用的指示器是通信复杂性，因为当任何算法达到并行可扩展的极限时，通信都会成为瓶颈。FFT 对 d 维分解的通信复杂性是 $O(P^{1/d})$ ，多网格的通信复杂性是 $O(\log P)$ 。我们最近证明了 FMM 的通信复杂性也是 $O(\log P)$ 。因此，FMM 和多网格都是通信最优的，FMM $O(\log P)$ 的证明也可以扩展到它的代数变体。

10.3 直接 N 体代码

本节将展示一个直接 N 体内核的例子，首先是简单的 C 语言版本，然后使用 SIMD 内置函数。简单的 N 体内核如图 10-1 所示。代码定义了 8 个数组用于描述主体的特征，数组

x 、 y 、 z 是坐标， m 是质量 / 电荷， p 是电势， ax 、 ay 、 az 是在每个方向的加速度。这里计算的方程是平滑拉普拉斯电势

```
#pragma simd
#pragma omp for
for (i=0; i<N; i++) {
    float pi = 0;
    float axi = 0;
    float ayi = 0;
    float azi = 0;
    float xi = x[i];
    float yi = y[i];
    float zi = z[i];
    for (j=0; j<N; j++) {
        float dx = x[j] - xi;
        float dy = y[j] - yi;
        float dz = z[j] - zi;
        float R2 = dx * dx + dy * dy + dz * dz + EPS2;
        float invR = 1.0f / sqrtf(R2);
        float invR3 = m[j] * invR * invR * invR;
        pi += m[j] * invR;
        axi += dx * invR3;
        ayi += dy * invR3;
        azi += dz * invR3;
    }
    p[i] = pi;
    ax[i] = axi;
    ay[i] = ayi;
    az[i] = azi;
}
```

图 10-1 代码清单——直接 N 体内核。目标（外层循环）被向量化

$$\phi_i = \sum_{j=1}^N \frac{m_j}{r_{ij}}$$

及加速度

$$a_i = \nabla \phi_i = - \sum_{j=1}^N \frac{m_j r_{ij}}{r_{ij}^3}$$

其中，

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \varepsilon^2}$$

是主体 i 和 j 之间的距离， ε 是平滑因子。

数组 x 、 y 、 z 、 m 是指定的值——在当前例子中，是 0 和 1 之间的随机数。这些值用来计算作用在所有 N 个主体上由所有 N 个主体感生的电压和加速度。这一过程产生了一个从 0 到 $N-1$ 的双循环，如图 10-1 所示。变量 i 的循环遍历目标主体， j 的循环遍历源主体。

本节使用图 10-1 的代码作为基准来看代码不改变时协处理器能取得的性能。同时借助 SIMD 和 OpenMP 两个编译指示语可以使用线程和 SIMD 向量来并行化外层循环。下一节将展示使用不同编译选项的性能结果。处理器和协处理器使用相同的代码。

协处理器有类似 SSE 和 AVX 的 512 位宽 SIMD 内置函数。因为这些内置函数指令直接对应到编译指令，编译器如何处理代码将比较明确。更具体地说，它将通知编译器明确地执行 *load*、*store*、*fmadd* 和 *rsqrt* 指令，且待向量化的循环也会明确指定。

直接 N 体内核的外层循环遍历目标主体，内层遍历源主体。使用 SIMD 内置函数可以指明向量化外层循环，方法是将 16 个数组元素放入 SIMD 寄存器，循环的跨度是 16，如图

10-2 所示。直接 N 体代码的大体结构没有改变，但所有的操作都写成 `_mm512` 的内置函数，所有的中间值都声明为 `_m512` 寄存器。如有需要，`fmadd` 指令将清晰指明。

```
#pragma omp for
for (i=0; i<N; i+=16) {
    _m512 pi = _mm512_setzero_ps();
    _m512 axi = _mm512_setzero_ps();
    _m512 ayi = _mm512_setzero_ps();
    _m512 azi = _mm512_setzero_ps();
    _m512 xi = _mm512_load_ps(x+i);
    _m512 yi = _mm512_load_ps(y+i);
    _m512 zi = _mm512_load_ps(z+i);
    for (j=0; j<N; j++) {
        _m512 xj = _mm512_set1_ps(x[j]);
        xj = _mm512_sub_ps(xj, xi);
        _m512 yj = _mm512_set1_ps(y[j]);
        yj = _mm512_sub_ps(yj, yi);
        _m512 zj = _mm512_set1_ps(z[j]);
        zj = _mm512_sub_ps(zj, zi);
        _m512 R2 = _mm512_set1_ps(EPS2);
        R2 = _mm512_fmadd_ps(xj, xj, R2);
        R2 = _mm512_fmadd_ps(yj, yj, R2);
        R2 = _mm512_fmadd_ps(zj, zj, R2);
        _m512 mj = _mm512_set1_ps(m[j]);
        _m512 invR = _mm512_rsqrt23_ps(R2);
        mj = _mm512_mul_ps(mj, invR);
        pi = _mm512_add_ps(pi, mj);
        invR = _mm512_mul_ps(invR, invR);
        invR = _mm512_mul_ps(invR, mj);
        axi = _mm512_fmadd_ps(xj, invR, axi);
        ayi = _mm512_fmadd_ps(yj, invR, ayi);
        azi = _mm512_fmadd_ps(zj, invR, azi);
    }
    _mm512_store_ps(p+i, pi);
    _mm512_store_ps(ax+i, axi);
    _mm512_store_ps(ay+i, ayi);
    _mm512_store_ps(az+i, azi);
}
```

图 10-2 代码清单——使用内置函数的直接 N 体内核。目标（外层循环）被向量化

向量化内层循环的一个替换形式如图 10-3 所示。代码与图 10-2 中基本相同，不同之处是跨度 16 现在在 j 循环中，且结尾处必须执行 `reduce-add` 操作，而不是简单的 `store` 操作。

```
#pragma omp for
for (i=0; i<N; i++) {
    _m512 pi = _mm512_setzero_ps();
    _m512 axi = _mm512_setzero_ps();

    _m512 ayi = _mm512_setzero_ps();
    _m512 azi = _mm512_setzero_ps();
    _m512 xi = _mm512_set1_ps(x[i]);
    _m512 yi = _mm512_set1_ps(y[i]);
    _m512 zi = _mm512_set1_ps(z[i]);
    for (j=0; j<N; j+=16) {
        _m512 xj = _mm512_load_ps(x+j);
        xj = _mm512_sub_ps(xj, xi);
        _m512 yj = _mm512_load_ps(y+j);
        yj = _mm512_sub_ps(yj, yi);
        _m512 zj = _mm512_load_ps(z+j);
        zj = _mm512_sub_ps(zj, zi);
        _m512 R2 = _mm512_set1_ps(EPS2);
        R2 = _mm512_fmadd_ps(xj, xj, R2);
        R2 = _mm512_fmadd_ps(yj, yj, R2);
        R2 = _mm512_fmadd_ps(zj, zj, R2);
        _m512 mj = _mm512_load_ps(m+j);
```

图 10-3 代码清单——使用内置函数的直接 N 体内核。源（内层循环）被向量化


```

    _mm512_invR = _mm512_rsqrt23_ps(R2);
    mj = _mm512_mul_ps(mj, invR);
    pi = _mm512_add_ps(pi, mj);
    invR = _mm512_mul_ps(invR, invR);
    invR = _mm512_mul_ps(invR, mj);
    axi = _mm512_fmadd_ps(xj, invR, axi);
    ayi = _mm512_fmadd_ps(yj, invR, ayi);
    azi = _mm512_fmadd_ps(zj, invR, azi);
}
p[i] = _mm512_reduce_add_ps(pi);
ax[i] = _mm512_reduce_add_ps(axi);
ay[i] = _mm512_reduce_add_ps(ayi);
az[i] = _mm512_reduce_add_ps(azi);
}

```

图 10-3 (续)

AVX 内置函数的代码与图 10-2 和图 10-3 所示的代码非常相似, 不同是“fmadd”要变成分开的“mul”和“add”操作, 图 10-3 中的“reduce: add”操作要变为“permute2f128”“add”和“hadd”的组合操作。

10.4 性能结果

本节将汇报图 10-1 和图 10-2 所示的直接 N 体内核在 Intel Xeon E5 v2 处理器和 Intel Xeon Phi 协处理器上的性能。测试使用 Intel MPSS 3.2.1 和本地模式的 Intel C++ Composer XE14.0.1 版本在 Intel Xeon Phi 7120P 协处理器, 以及使用 Intel C++ Composer XE13.0.1 版本的两个 Intel Xeon E5-2680 v2 (Ivy Bridge) 上执行。执行时 $N = 65\,536$, 每对主体的性能为 20FLOPS。处理器上使用的编译选项是“-mavx -openmp -O3”, 协处理器使用的是“-mmic -openmp -fimf-domain-exclusion = 15 -O3”。图 10-4 显示了使用指令和内置函数时处理器和协处理器的比较结果。

本实验在双插槽处理器上使用 40 个线程, 在协处理器上使用 244 个线程。

协处理器上线程的可扩展性非常好, 图 10-5 将展示更多细节。图例“-mavx+pragma”“_mm256”“-mmic+pragma”“_mm512”分别表示主处理器使用指令、主处理器使用内置函数、协处理器使用指令和协处理器使用内置函数。在处理器上, 基于内置函数的版本比基于指令的版本性能要好, 但在协处理器上, 基于指令的版本运行更快。这两种基于指令和基于内置函数的版本都比之前的直接 N 体版本运行在协处理器上的性能好。

图 10-5 显示当使用不同线程数量时直接 N 体内核的强可扩展性。Intel Xeon Phi 7120P 协处理器有 61 个内核, 每个内核 4 个线程, 所以线程总数达到 244。本实验通过将“KMP_AFFINITY”变为 compact、balanced 和 scatter 来测试线程关联的效果。scatter 和 balanced 选项显示了直到 61 个内核都具有理想的可扩展性, 每个内核上运行 1 个线程。compact 选项当一个内核上 4 个线程都运行时得到理想的可扩展性。这说明处理器内核的可扩展性非常好, 但一个内核内 4 个线程的可扩展性并不好。最终, 当 244 个线程都使用时, 不同

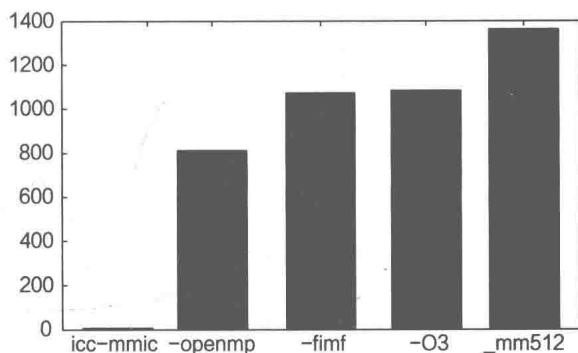


图 10-4 使用指令和内置函数的直接 N 体内核在 Ivy Bridge 处理器 (avx/mm256) 和 Intel Xeon Phi 协处理器 (mic/mm512) 上的单精度 GFLOPS

“KMP_AFFINITY” 选项不会导致任何区别。

到现在为止，图 10-4 和图 10-5 所示实验针对的问题大小是确定的 $N = 65\,536$ 。图 10-6 显示直接 N 体内核在不同问题大小时的单精度 GFLOPS。在图例中，“ I intrinsics” “ J intrinsics” “ I pragma” “ J pragma” 分别表示使用内置函数向量化目标循环、使用内置函数向量化源循环、使用指令向量化目标循环和使用指令向量化源循环。误差条显示了对于每种情况 256 次运行产生的标准偏差，单位是 GFLOPS。这里显示误差条是由于一些情况下运行时的偏差非常大。从图中首先注意到源向量化版本在 $N = 2^{14} = 16\,384$ 时达到峰值。使用 SIMD 内置函数和指令 `#pragma simd` 能明确指示编译器向量化外层循环，使得当 $N > 16\,384$ 时产生更高的性能。对源向量化使用 SIMD 内置函数运行更快，而对目标向量化，使用基于指令的版本运行更快。向量化内层还是外层循环的最优选择取决于问题大小。

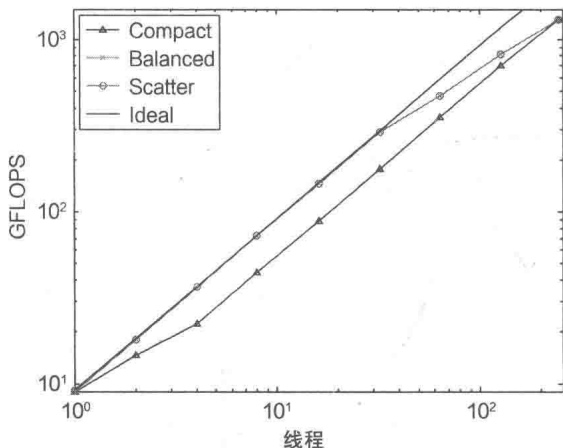


图 10-5 不使用内置函数的直接 N 体内核在 Intel Xeon Phi 协处理器在不同内核数时的单精度 GFLOPS

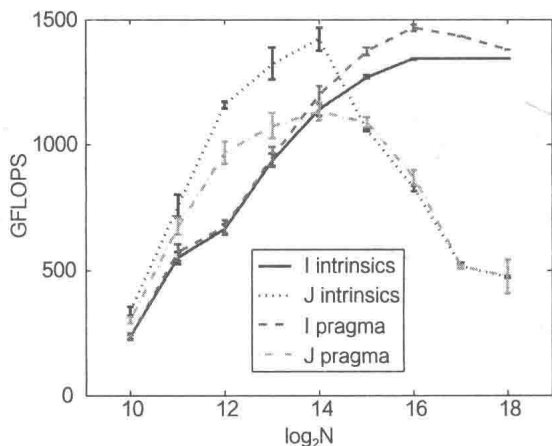


图 10-6 对不同问题大小的直接 N 体内核使用或不使用内置函数，对 i 的外层循环或 j 的内层循环进行向量化，在 Intel Xeon Phi 协处理器的单精度 GFLOPS

10.5 总结

本章描述了如何在协处理器优化直接 N 体内核。仅通过简单地使用编译器选项 “`icc -mmic -openmp -fimf-domain-exclusion=15`” 及向 CPU 的原始 C 代码添加 “`#pragma simd`” 可达到 1.5TFLOPS 单精度浮点性能。使用 OpenMP 和 `simd` 指令有益于处理器，程序执行过程接近完美的强可扩展性，而内核内部线程的可扩展效率较低。当使用全部 244 个线程时，使用 “KMP_AFFINITY” 没有任何影响。

通过使用 `_mm512` 内置函数，即使对较小的问题规模也能够取得 1.4TFLOPS 的性能。性能对问题大小有很强的依赖性，且这种关系不是单调的。直到一定的问题规模，向量化内层循环的性能更好，但当 $N > 16\,384$ 时，向量化外层循环的性能更好。通过使用可移植且通用的 `#pragma simd` 指令或者更详细的 `_mm512` 内置函数可以向量化外层循环。

10.6 更多信息

下面是本章推荐的一些额外阅读材料：

- Mini N-body kernels, <https://github.com/harrism/mini-nbody>.
- Test-driving Intel Xeon Phi Coprocessors with Basic N-body Simulation.
- <http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx>.
- 本章及其他章源代码下载地址 <http://lotsofcores.com>。
- Aarseth, S., 1963. Dynamical evolution of clusters of galaxies. *Mon. Not. R. Astron. Soc.* 126, 223-255.
- Alder, B.J., 1957. Phase transition for a hard sphere system. *J. Chem. Phys.* 27, 1208-1209.
- Sambavaram, S.R., Sarin, V., Sameh, A., Grama, A., 2003. Multipole-based preconditioners for large sparse linear systems. *Parallel Comput.* 29, 1261-1273.
- Barnes, J., Hut, P., 1986. $O(N \log N)$ force-calculation algorithm. *Nature* 324, 446-449.
- Greengard, L., Rokhlin, V., 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73, 325-348.
- Johnson, S., Frigo, M., 2007. A modified split-radix FFT with fewer arithmetic operations. *IEEE Trans. Signal Process.* 55, 111-119.
- Gholami, D., Malhotra, H., Sundar, G. Biros, FFT, FMM, or multigrid? A comparative study of state-of-the-art Poisson solvers. *SIAM J. Scient. Comput.*, submitted, <http://users.ices.utexas.edu/~hari/files/pubs/sisc14.pdf>.
- Greengard, L., Gueyffier, D., Martinsson, P.G., Rokhlin, V., 2009. Fast direct solvers for integral equations in complex three-dimensional domains. *Acta Numerica* 18, 243-275.
- Gu, M., Eisenstat, S.C., 1996. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM J. Scient. Comput.* 17, 848-869.
- Grasedyck, L., Hackbusch, W., 2003. Construction and arithmetics of H-matrices. *Computing* 70, 295-334.
- Rjasanow, S., 2002. Adaptive cross approximation of dense matrices. In *International Association for Boundary Element Methods*, UT Austin, TX, USA, May 28-30, 2002.
- Yokota, R., Turkiyyah, G., Keyes, D., 2014. Communication complexity of the FMM and its algebraic variants, arXiv:1406.1974.
- Vladimirov, V., Karpusenko, Test-driving Intel Xeon Phi Coprocessors with Basic N-body Simulation, Colfax International, 2013, <http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx>.

使用 OpenMP 4.0 实现动态负载均衡

Gilles Givario, Michael Lysaght

爱尔兰, ICHEC

多年来,在高性能计算社区,OpenMP 是在共享内存多处理器上进行编程的可选模型。虽然 OpenMP 号称易于编程且支持增量并行,但是没有经过优化的 OpenMP 程序的性能往往不高。事实上,同其他编程模型(如 MPI)一样,使用 OpenMP 编程也需要关注算法和硬件细节。

最新发布的 OpenMP 4.0 标准更需要这样。OpenMP 4.0 支持“卸载”模式,能够充分发挥各种新兴众核处理器(如 Intel Xeon Phi 协处理器)的优势。一般情况下,在给定的处理器(如 Intel Xeon 多核处理器)上进行 OpenMP 编程会相对简单。然而,要充分利用异构系统上所有的可用计算资源(如“Stampede”或者“天河 2 号”超级计算机的每个节点既使用了 Intel Xeon 处理器也使用了 Intel Xeon Phi 协处理器)要困难得多。例如,确定主机端和设备端并行数据传输的最佳方法就会提出很多挑战。在此之上,为保证每个设备和主机端的负载均衡(哪些任务需要卸载到设备端处理,哪些任务依然由主机端处理),OpenMP 编程又增加了一个额外的复杂度。本章将通过一个简单的 N 体算法,指导读者完成从串行程序到深度并行政务的开发。该并行程序能够充分利用异构计算系统中的所有可用处理单元,并且能够自动为每个设备动态分配最佳工作负载,从而实现最高加速比。本章描述的动态负载均衡方法完全使用 OpenMP 4.0 标准编写,这使得这个方法不仅具有完美的可移植性,而且足够通用,从而使得该方法能够简单地应用到其他广泛的计算问题(如天体物理学和分子动力学)中。无论对于代码开发者,还是在混淆来源或者牺牲性能前提下急于发布代码二进制版本的软件厂商来说,这都是非常有趣的。

11.1 最大化硬件利用率

本章剩余的几节将使用众核加速器领域的传统术语。在本章中,CPU、处理器或者主机系统一般指的是 Intel Xeon 处理器(CPU)。协处理器、设备或者加速器一般指的是 Intel Xeon Phi 协处理器。结合上下文,我们也使用“计算设备”表示协处理器和 CPU。

Intel Xeon Phi 协处理器能够使用三种主要的编程范式进行编程:“本机”模式、“对称”模式以及“卸载”模式。本机模式是通过最小的代码改动实现已有软件在 Intel Xeon Phi 协处理器上运行的最快方式。与通用图像处理器(GPGPU)不同,Intel Xeon Phi 协处理器能够执行本机应用程序,该程序不需要运行在 CPU 上的主机进程,直接运行在协处理器的操作系统上。在这种模式下,每个协处理器都可以看作一个独立的众核多路对称计算设备。当然,以本机模式在协处理器上运行一份代码,需要对这份代码进行移植。然而,得益于 Intel 处理器和协处理器以及对应软件栈的相似性,代码移植相对简单。相对于标准多核处理器,要充分利用协处理器上更多的内核(或者更宽的向量单元),可能需要更多的工作。但两者可以使用相同的编程模型。这就是当用户使用 Intel Xeon Phi 架构时,会首先使

用本机模式作为移植工作的第一步的原因。然而，在协处理器上使用本机模式有两点劣势：1) 只能利用一个 Intel Xeon Phi 协处理器；2) 浪费 CPU 上的可用计算资源。

将已经能够并行运行在分布式内存系统上的代码移植到协处理器上，对称模式是非常合适的选择。之所以称为对称模式，是因为该模式允许代码同时利用给定集群上的处理器和协处理器的所有可用计算资源。在这种情况下，所有的计算设备能够并行访问。虽然在对称模式下将代码移植到基于协处理器的集群上所需要的工作和使用本机模式一样简单，但是在异构计算系统上实现负载均衡是非常具有挑战性的。

当使用卸载模式时，应用程序在主机系统（即使用 CPU）上加载启动，数据的初始化也在主机端进行。随后，应用程序使用 PCIe 总线将特定的可执行代码及所需要的数据推送（卸载）到设备端执行。设备端执行完毕后，将计算结果返回主机端。对于 Intel Xeon Phi 协处理器，程序员能够以编译制导的方式实现卸载初始化。卸载模式不但可以用在独立的机器上，而且可用在集群上。当在集群上使用卸载模式时，集群的每一台机器将会开启一个或者多个 MPI 进程，然后每个进程各自执行卸载操作。

在三种编程模式中，尽管卸载模式常常是最耗费精力的，但其潜在回报也是最高的。尤其是该模式下：

- 允许一份代码同时使用主机端和设备端的计算资源。
- 避免了在大规模代码中特别具有挑战性的 MPI 层次上负载均衡的复杂性。
- 编写的代码适用于任何可预见的 Intel Xeon 和 Intel Xeon Phi 产品。

然而，以有效的方式充分开发和利用卸载模式的优点是非常具有挑战性的。例如，卸载模式需要同时利用主机和设备的所有计算单元，这样才能利用两者潜在的“组合”计算能力，而不是在两个计算设备间切换，但要实现这个功能不容易（见图 11-1）。除此之外，实现计算设备间的最佳负载均衡，从而使所有计算设备同时到达同步点，消除等待某一计算设备到达造成的时间开销，也是非常具有挑战性的。

本章描述了如何在 Intel Xeon 处理器和任意多个 Intel Xeon Phi 协处理器间实现最佳负载均衡。同时，本章使用最新发布的 OpenMP 4.0 标准，以充分利用两者可使用统一编程模型的优点。该新标准允许我们仅仅使用几条编译制导指令，就可以通过一种可移植及优雅的方式，使用完全相同的代码库来充分利用异构计算平台的所有潜在计算能力。实现这一目标需要的唯一一个 OpenMP 4.0 新特性就是新的 SIMD 指令。全部代码的性能以及卸载指令，我们都将在 Intel Xeon Phi 协处理器和 Intel Xeon 处理器上应用和分析。

本章将使用在天文物理学应用程序中常用的一个经典 N 体算法来充分说明异构系统程序的整体开发方法。类似的算法也用在分子动力学应用程序中，在这个应用中重力被替换为库仑力。事实上，需要强调的是，本章描述的负载均衡算法和具体的计算内核是相互独立的。因此，这个方法也可应用到其他迭代算法中。



图 11-1 如果将相同的工作负载分配给工作能力不同的两个协同工作者，工作负载的分配明显困难

11.2 N 体内核

为帮助说明该方法，我们假设要模拟太空中非相对论粒子的自由演化，如卫星、行星、恒星、星云等。根据牛顿第二定律，每个粒子所受到的总作用力等于其质量和加速度的乘积。同时，根据牛顿万有引力定律，任意两个粒子间的相互吸引力与它们质量的乘积成正比，与它们距离的平方成反比。对于 N 个粒子的集合，牛顿第二定律可用如下公式表达：

$$\forall i \in \{1, \dots, N\} m_i \cdot \vec{a}_i = G \sum_{i \neq j} \frac{m_i \cdot m_j}{d_{ij}^2} \vec{u}_{ji} \quad (11-1)$$

其中， m_i 为粒子 i 的质量， \vec{a}_i 为粒子 i 的加速度， G 为万有引力常数， d_{ij} 为粒子 i 和粒子 j 间的距离， \vec{u}_{ij} 为粒子 i 和粒子 j 间的单一向量。式 (11-1) 可重写为：

$$\forall i \in \{1, \dots, N\} \frac{d\vec{v}_i}{dt} = G \sum_{i \neq j} \frac{\vec{x}_j - \vec{x}_i}{d_{ij}^3} m_j \quad (11-2)$$

其中， \vec{v}_i 为粒子速度， \vec{x}_i 为粒子位置， t 为时间。

根据式 (11-2)，可以非常容易写出 N 体万有引力内核函数，如图 11-2 所示。此外，我们也想评估浮点精度对性能的影响，因此引入一个通用“实”类型，该类型根据宏定义取值 `double` 或 `float`。

```
#ifndef DOUBLE
    typedef double real;
    #define Sqrt sqrt
#else
    typedef float real;
    #define Sqrt sqrtf
#endif

real *x, *y, *z, *vx, *vy, *vz, *m;
const real G = 6.67384e-11;

void Newton( size_t n, real dt ) {
    const real dtG = dt * G;
    for ( size_t i = 0; i < n; ++i ) {
        real dvx = 0, dvy = 0, dvz = 0;
        for ( size_t j = 0; j < n; ++j ) {
            if ( j != i ) {
                real dx = x[j] - x[i], dy = y[j] - y[i],
                    dz = z[j] - z[i];
                real dist2 = dx*dx + dy*dy + dz*dz;
                real mjOverDist3 = m[j] /
                    (dist2 * Sqrt( dist2 ));
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
        }
        vx[i] += dvx * dtG;
        vy[i] += dvy * dtG;
        vz[i] += dvz * dtG;
    }
    for ( size_t i = 0; i < n; ++i ) {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;
        z[i] += vz[i] * dt;
    }
}
```

图 11-2 初始 N 体万有引力内核函数

由于次优的初始内核函数，为保证性能比较没有任何偏见，我们使用 OpenMP 指令并行优化 Newton 函数：

1. 使用 OpenMP 的 “parallel” 结构包含整个函数体。
2. 在两个遍历 i 的 for 循环上分别增加 OpenMP “for” 指令，将循环迭代分配到多个 OpenMP 线程中。
3. 在 for 结构上增加 “schedule (auto)” 指令，将任务调度委派给编译器，所以这不是我们关注的重点。
4. 在两个遍历 j 的 for 循环上分别增加 OpenMP “simd” 指令，使编译器在 “ j ” 上完成自动向量化工作。
5. 为消除变量 i 和 j 进行比较操作的开销，这个语句可能会阻碍循环的向量化，将这个循环拆分成两个，一个从 0 到 i ，另一个从 $i+1$ 到 n 。

通过这些修改，最新的内核函数如图 11-3 所示。

```
void Newton( size_t n, real dt ) {
    const real dtG = dt * G;
    #pragma omp parallel
    {
        #pragma omp for schedule( auto )
        for ( size_t i = 0; i < n; ++i ) {
            real dvx = 0, dvy = 0, dvz = 0;
            #pragma omp simd
            for ( size_t j = 0; j < i; ++j ) {
                real dx = x[j] - x[i], dy = y[j] - y[i],
                    dz = z[j] - z[i];
                real dist2 = dx*dx + dy*dy + dz*dz;
                real mjOverDist3 = m[j] /
                    (dist2 * Sqrt( dist2 ));
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
            #pragma omp simd
            for ( size_t j = i+1; j < n; ++j ) {
                real dx = x[j] - x[i], dy = y[j] - y[i],
                    dz = z[j] - z[i];
                real dist2 = dx*dx + dy*dy + dz*dz;
                real mjOverDist3 = m[j] /
                    (dist2 * Sqrt( dist2 ));
                dvx += mjOverDist3 * dx;
                dvy += mjOverDist3 * dy;
                dvz += mjOverDist3 * dz;
            }
            vx[i] += dvx * dtG;
            vy[i] += dvy * dtG;
            vz[i] += dvz * dtG;
        }
        #pragma omp for simd schedule( auto )
        for ( size_t i = 0; i < n; ++i ) {
            x[i] += vx[i] * dt;
            y[i] += vy[i] * dt;
            z[i] += vz[i] * dt;
        }
    }
}
```

图 11-3 优化后的 N 体万有引力内核函数

如图 11-4 所示，我们分别在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器（使用本机模式）上对 N 体内核进行测试。为了实现这一点，我们额外写了一个程序，该程序用于分配存

储粒子位置和速度的数组，并使用随机初始条件对这两个数组进行了初始化。随后调用 Newton 函数 N 次 (N 为给定的迭代次数)。

对于此处讨论的模拟过程，我们设定 $n = 50\,000$ 个粒子，时间步 $dt = 0.01$ ，并允许系统给定 1s 的“模拟”时间，在这里代表 100 次迭代。我们使用 Intel 编译器分别在处理器和协处理器上编译了这个程序（注意，此时在协处理器上采用本机模式进行编译），随后将编译好的程序运行在开发平台上。[⊖]

图 11-5 展示了程序在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器两个计算平台上的性能及良好的可扩展性。这里值得注意的是，该程序不仅充分利用了协处理器上硬件线程的性能（这是可以预料的），还充分利用了处理器硬件线程的性能。其中，在处理器平台上，我们通过使用超线程技术获得了 21% 的性能提升。最终，我们在 Intel 运行时库默认的线程数目上获得了最短运行时间，也就是说，我们并没有设置 `OMP_NUM_THREADS` 去获取最佳性能。这些条件将在本章的剩余部分继续使用。

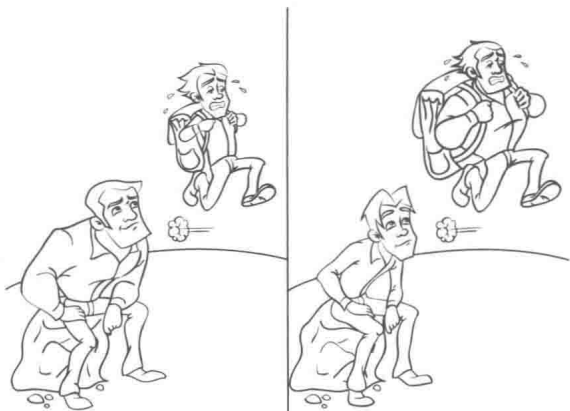


图 11-4 不平等的工作分配是低效的。我们当然可以非常容易地将所有工作安排给处理器或者协处理器，此时会有其中一个计算设备（协处理器或者处理器）处于空闲状态。这些浪费计算资源的情况就是本章描述的负载均衡的动机

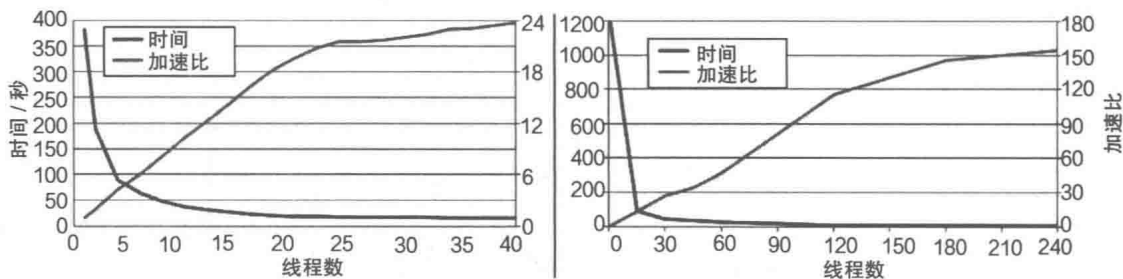


图 11-5 N 体内核在两个处理器（左）和一个协处理器（右）上的良好可扩展性

图 11-6 展示了 N 体内核在主机端和设备端的绝对性能（单精度和双精度）。从中可以看出，相对于在两个 10 核 Ivy Bridge Intel Xeon 处理器上的性能，我们在 Intel Xeon Phi 协处理器上分别实现了单精度 2.17 倍和双精度 3.19 倍的性能提升。如果我们认为平方根是一个正常的浮点运算，那么在 Intel Xeon 处理器上达到了 125 GFLOPS 的性能（单精度），在 Intel Xeon Phi 协处理器上达

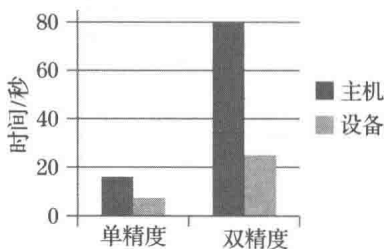


图 11-6 N 体内核在两个（主机）处理器和一个（设备）协处理器上的完成时间（秒）。注意，时间轴越短代表性能越好

⊖ 计算节点由两个 10 核的 Intel Xeon E5-2660 v2 处理器和两个 60 核的 Intel Xeon Phi 5110P 协处理器构成。

到了 540GFLOPS 的性能（单精度）。这意味着经过简单优化的 N 体内核就利用了处理器峰值性能的 36%，协处理器峰值性能的 27%。考虑到我们在优化内核中的投入，这是一个非常可观的成绩。

从这个坚实的基础出发，现在我们可以非常自信地开始我们的旅程，目标是开发一段完全集成的代码。

11.3 卸载版本

我们的目标是将上一节讨论的 N 体代码转化为卸载模式：在主机端（处理器）运行程序并将计算和数据推送给设备端（协处理器），如图 11-7 所示。为实现这一点，只需要添加三条额外的 OpenMP 指令：

- 使用 “#pragma omp declare target” 与相应的 “#pragma omp end declare target” 指令包围所有需要在协处理器上用到的变量和函数定义。这将强制编译器生成两个版本的代码：一个用于处理器，一个用于协处理器。这两个版本的代码将链接为一份二进制代码。
- Newton 函数中，在 “#pragma omp parallel” 指令前添加 “#pragma omp target” 指令。这将指示编译器将计算卸载到计算平台上的第一个 Intel Xeon Phi 协处理器设备上。
- 最后，如图 11-8 所示，在调用 Newton 函数的主循环上方添加 “#pragma omp target data” 指令。这条指令将指示编译器在设备端为数组序列分配内存，并在执行进一步的操作前，将初始值从主机端复制到设备端。最终，在主循环的出口处从设备中获得位于 “map (tofrom:)” 列表中的数组的值。



图 11-7 现在主机端和设备端相互了解了对方，可以共同承担工作负载了

```
#pragma omp target data\
    map( tofrom: x[0:n], y[0:n], z[0:n] ) \
    map( to: vx[0:n], vy[0:n], vz[0:n], m[0:n] )
for ( int it = 0; it < 100; ++it ) {
    Newton( n, 0.01 );
}
```

图 11-8 调用 N 体内核时使用的数据卸载 OpenMP 指令

现在我们可以运行 N 体内核的卸载版本，并同处理器的本机版本和协处理器的本机版本进行性能比较。图 11-9 展示了比较结果。

所以，我们有了一份可以同往常一样运行的代码（不需要登录 Intel Xeon Phi 协处理器）。同它的本机版本相比，这份代码只有一点点的性能损失。现在是时候以“协作”的方式使用异构系统中的所有可用处理器和协处理器资源，以充分发挥系统的计算潜力。

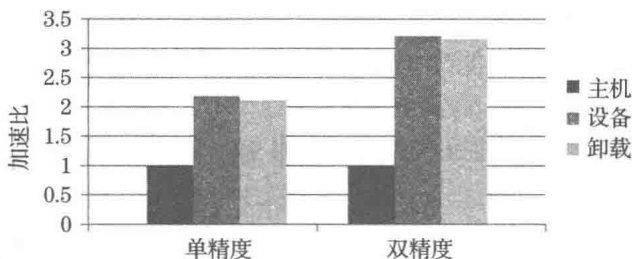


图 11-9 将 N 体内核在两个 Intel Xeon E5-2660 v2 Ivy Bridge 处理器上的性能归一化为 1 后，三个版本的性能比较（越高越好）：本机处理器版本、本机协处理器版本、卸载协处理器版本

11.4 第一个处理器与协处理器协作版本

我们的目标是找到一种方法：

1. 将部分工作负载卸载到 Intel Xeon Phi 协处理器上，同时余下的工作负载继续在 Intel Xeon 处理器（主机）上进行处理和计算。

2. 确保设备端和主机端的负载均衡，即将适量工作负载提交给设备端处理，保证设备端和主机端可并行执行相同时间。要实现这点的关键是确保一个计算设备在等待其他计算设备完成各自任务时，没有时间浪费（见图 11-10）。

然而，由于在计算过程中的每个时间步长，系统都需要完整的关于粒子速度和位置的最新视图，所以这种主机-设备分配工作、协同工作的方式会导致额外的数据传输。这种额外的数据传输不但需要精心管理以确保代码的正确性，而且会对性能产生明显影响。



图 11-10 根据不同处理器的计算能力对工作负载进行分配。使处理器和协处理器完成任务的时间尽可能相同，以此减少整个系统的空闲时间

如何解决这些问题呢？第一步，创建两个线程，以允许主机和设备并行运行，每个线程管理其中一部分。这只需要在循环前加上“`#pragma omp parallel num_threads(2)`”指令就可实现。然而，这会对主机代码产生一定的影响：

- 要确保计算在主机上还能够并行执行，我们需要允许嵌套的 OpenMP 并行性。幸运的是，这只需要在代码的初始化阶段增加对“`omp_set_nested(true)`”的调用即可实现。
- 需要主机的一个专门 OpenMP 线程管理设备，这就意味着减少了主机端可用于计算的线程数目。虽然这对拥有大量内核的主机（如本章使用的计算平台）的影响有限，但确实是一个不能忽略的要素。

在完成第一步后，需要在主机和设备间分发任务。这个工作的原理非常简单，并且只需要稍微修改 Newton 函数：增加两个输入参数。这两个输入参数定义了基于 i 循环的迭代范围。在 Newton 函数内部，我们使用这两个额外参数来限制基于 i 循环的迭代次数。除此之外，我们在“`#pragma omp target`”中增加一条额外的语句“`if (n0==0)`”来选择性地将计算任务卸载到设备上。在这里， $n0$ 为增加的第一个额外参数，表示循环迭代的起始值。通过这种方式，基于 i 的循环只将前一组迭代卸载到设备上执行，其他迭代依然在主机上执行。

此时，我们需要在每个时间步长结束时保证主机和设备上数据的同步。为此，我们使用“`#pragma omp target update`”指令将数据发送到设备或者从设备取回数据。为保证

数据的一致性，我们需要在更新数据的前后增加栅栏（barrier）。更进一步，因为这些数据交换必须由一个线程来执行，所以我们增加“#pragma omp single”指令。这条语句在其最后有隐式栅栏，因此可消除对应的显式调用。完成这些操作后，除了定义计算任务卸载到设备与留在主机的比例，我们可以运行代码。接下来，我们将描述这部分工作是如何进行的。

首先，假设主机和设备会按照一定的性能来处理计算任务。虽然最初我们并不知道计算设备的真实性能，但可以假定它是固定的。计算设备的性能可以用每秒处理的 Newton 函数内部的外层 i 循环的迭代次数来表示。为实现计算设备间的负载均衡，我们在每个时间步长结束后，根据实测性能动态调整迭代次数，以使 Newton 函数在主机和设备上的完成时间相同。这就得出了下面的公式：

$$r' = \frac{t_h r}{t_h r + t_d (1 - r)} \quad (11-3)$$

式中， r 表示上一次迭代时分配到设备上任务的比例， r' 表示下一次迭代时的新比例。 t_h 表示主机完成任务的时间， t_d 表示设备完成任务的时间。

根据上面描述的公式，我们通过使用上一次迭代两个计算设备的实际运行时间，计算每个时间步长卸载到设备的最佳任务量。我们将分配到两个计算设备的任务比例初始化为 0.5，然后由系统自动进行调整。

图 11-11 为主函数的代码，图 11-12 为该版本代码的性能。

```
omp_set_nested( true );
double ratio = 0.5;
double tth[2];
#pragma omp target data \
    map( to: x[0:n], y[0:n], z[0:n], \
        vx[0:n], vy[0:n], vz[0:n], m[0:n] )
{
    #pragma omp parallel num_threads( 2 )
    {
        const int tid = omp_get_thread_num();
        for ( int it = 0; it < 100; ++it ) {
            size_t lim = n * ratio;
            size_t l = n - lim;
            double tt = omp_get_wtime();
            Newton( lim*tid, lim + l*tid, n, 0.01 );
            tth[tid] = omp_get_wtime() - tt;
            #pragma omp barrier
            #pragma omp single
            {
                #pragma omp target update \
                    from( x[0:lim], y[0:lim], z[0:lim], \
                        vx[0:lim], vy[0:lim], vz[0:lim] )
                #pragma omp target update \
                    to( x[lim:l], y[lim:l], z[lim:l], \
                        vx[lim:l], vy[lim:l], vz[lim:l] )
                ratio = ratio*tth[1] /
                    (ratio*tth[1] + (1 - ratio)*tth[0]);
            }
        }
    }
}
```

图 11-11 主机加一个设备对 N 体内核的均衡调用

图 11-13 展示了每个时间步长设备端工作负载的分配情况。我们可以看到，经过一个短时间的“热身”之后，分配到设备端的工作负载比例大约为 0.7（单精度）和 0.76（双精度），这和之前测试的一致。

这个版本代码的最后一部分说明，数据更新由一个线程顺序执行。当然，数据更新新任务可以非常容易地并行实现，比如：第一个线程负责 $x[]$ 、 $y[]$ 和 $z[]$ 的数据更新，第二个线程负责 $vx[]$ 、 $vy[]$ 和 $vz[]$ 的数据更新。这将会充分利用连接 Intel Xeon Phi 协处理器和主机的 PCI 总线的双向能力。然而，在该例子中，与增加代码的复杂度相比，这种优化方法获得的性能提升实在微不足道。尽管如此，在代码的下载包中也提供了这个改进版本的代码。

11.5 多协处理器版本

假设本章采用的计算平台至少配备一个 Intel Xeon Phi 协处理器。然而，如果该计算平台上有多个可用设备，目前的代码只能利用其中一个。现在，我们解决在代码中充分利用任意多个设备的问题。与之前一样，我们将尽可能透明并有效地利用协处理器（如图 11-14 所示）。为实现该目标，我们将使用 OpenMP 4.0 的两个额外特性。

- 运行时库函数“`omp_get_num_devices()`”可以获得当前计算系统中可用的设备数目。
- 在代码中的“`#pragma omp target`”语句后面添加额外的“`device (dev)`”语句，定义要使用的设备索引。

有了这两个新特性，我们就可以实现代码的多功能目标：可以查找当前系统中可用的设备数量，并根据定义的条件访问其中的任意一个。但仍然需要理解，如何改变代码才能分配和传输数组到任意数量的设备上。如果我们简单地把之前代码版本用于任意数目的设备，主要代码如图 11-15 所示。

但从图 11-15 所示的代码可以看出，将并行结构包含进任意数量的“`target data`”指令中（一个设备一个）非常困难，甚至是不可能的。然而，我们只需要简单地交换“`target data`”和“`parallel`”两条指令的顺序，这个问题就变得非常简单。我们没有试图让主线程管理所有设备，而是采用多线程方式，让每一个线程至多只管理一个设备的

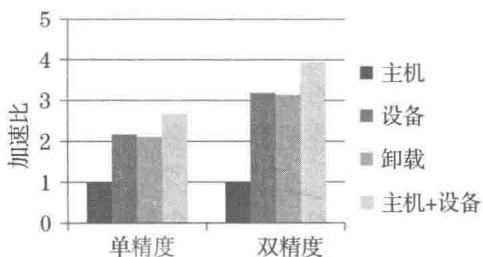


图 11-12 相较于在两个 Intel Xeon E5-2660 v2 Ivy Bridge 处理器上的性能，各种版本的 N 体内核的加速比（越高越好）

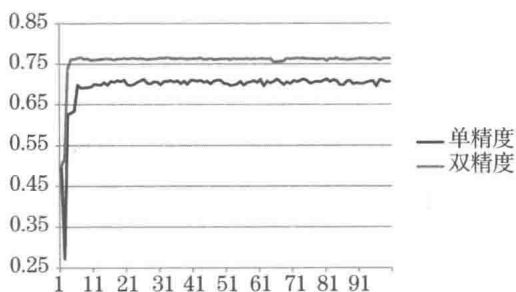


图 11-13 卸载到 Intel Xeon Phi 协处理器上的工作负载比例变化情况



图 11-14 可加入更多的“肌肉”以处理更多的任务。一旦实现了负载均衡，就可以增加更多的处理器和协处理器来共同完成计算任务

数据分配和传输。这样，代码就变为如图 11-16 所示。

```
const int dev = omp_get_num_devices();
#pragma omp target data device( ?? ) \
    map( to: x[0:n], y[0:n], z[0:n], m[0:n], \
        vx[0:n], vy[0:n], vz[0:n] )
{
    #pragma omp parallel num_threads( dev + 1 )
    {
        const int tid = omp_get_thread_num();
        for ( int it = 0; it < 100; ++it ) {
            ...
        }
    }
}
```

图 11-15 一个不可行的多设备版本

```
const int dev = omp_get_num_devices();
#pragma omp parallel num_threads( dev + 1 )
{
    int tid = omp_get_thread_num();
    #pragma omp target data device( tid ) if( tid < dev ) \
        map( to: x[0:n], y[0:n], z[0:n], m[0:n], \
            vx[0:n], vy[0:n], vz[0:n] )
    {
        for ( int it = 0; it < 100; ++it ) {
            ...
        }
    }
}
```

图 11-16 一个可行的多设备版本

现在，我们只需要根据可用的计算设备计算工作负载的均衡性，并且在每个时间步长之后调整数据更新，就可以完成代码。计算工作负载分配比例和之前描述的一样：假设每个计算设备都在额定的速度下运行。实现所有计算设备在相同时间内完成负载计算的预期目标会产生如下公式：

$$\forall i, j \in \{0, \dots, \text{dev}\} \frac{r'_i}{r_i} t_i = \frac{r'_j}{r_j} t_j \quad (11-4)$$

式中， r 和 t 分别代表工作负载分配比例和时间。 i 和 j 为设备索引， r' 为下次迭代的工作负载分配比例。我们知道：

$$\sum_{i=0}^{\text{dev}} r'_i = 1 \quad (11-5)$$

随后我们可以推断：

$$\forall j \in \{0, \dots, \text{dev}\} r'_j = \frac{r_j}{t_j \sum \frac{r_i}{t_i}} \quad (11-6)$$

因为我们要计算的是基于 i 循环的起始索引（数组的位移），所以可以使用图 11-17 所示的 `computeDisplacements` 函数实现这个功能。考虑到每个时间步长之后的数据更新，我们必须注意如下两点：

- 因为会有多个设备将其计算数据和主机进行同步更新，所以对于给定的数组，一次只允许一个设备的数据更新会带来更好的性能（尽管不是必需的）。这就是我们将“`update from`”指令包含进“`critical`”的原因。除此之外，我们还会增加一个“栅栏”，以保证主机数据在重新发送给设备之前已经全部更新完毕。对于前面的代码版本，当然可以充分利用需要更新多个数组来保持这个阶段的并行性，然而，

与获得的性能提升相比，引入的复杂性会使结果适得其反。

```
void computeDisplacements( size_t *displ,
                          const double *tth, int dev ) {
    double sumLengthOverT = 0;
    size_t length[dev+1];
    for ( int i = 0; i < dev+1; ++i )
        length[i] = displ[i+1] - displ[i];
    for ( int i = 0; i < dev+1; ++i )
        sumLengthOverT += length[i] / tth[i];
    for ( int i = 0; i < dev; ++i )
        displ[i+1] = displ[i] + round( (displ[dev+1] * length[i]) /
                                       (tth[i] * sumLengthOverT) );
}
```

图 11-17 在每个时间步长之后计算新的位移，在这里将 displ[0] 初始化为 0，将 displ[dev+1] 初始化为 n

- 由于每个设备都有原先数据的最新版本，所以只需要从主机获取额外增加的数据即可。这最多需要两个数组段就可以完成。同时，因为这里不存在数据一致性的问题，所以向所有设备发送数据可以并行执行。

图 11-18 展示了最终的循环。现在可以在包含两个 Intel Xeon Phi 协处理器设备的机器上运行该代码。最终版本代码的性能与之前版本的性能对比如图 11-19 所示。

```
for ( int it = 0; it < 100; ++it ) {
    size_t s = displ[tid], l = displ[tid+1] - displ[tid];
    double tt = omp_get_wtime();
    Newton( n, s, l, 0.01, tid, dev );
    tth[tid] = omp_get_wtime() - tt;
    #pragma omp critical
    {
        #pragma omp target update device( tid ) \
            if( tid < dev ) \
            from( x[s:l], y[s:l], z[s:l], \
                vx[s:l], vy[s:l], vz[s:l] )
    }
    #pragma omp barrier
    #pragma omp target update device( tid ) \
        if( tid < dev ) \
        to( x[0:s], y[0:s], z[0:s], \
            vx[0:s], vy[0:s], vz[0:s] )
    size_t s1 = s+1, l1 = n-s-1;
    #pragma omp target update device( tid ) \
        if( tid < dev ) \
        to( x[s1:l1], y[s1:l1], z[s1:l1], \
            vx[s1:l1], vy[s1:l1], vz[s1:l1] )
    #pragma omp single
        computeDisplacements( displ, tth, dev );
}
```

图 11-18 最终的主时间循环

图 11-20 展示了在开发平台上一个 Intel Xeon 处理器和两个 Intel Xeon Phi 协处理器是如何动态保持工作负载（粒子数比例）均衡的。从图中可以非常清晰地看出，经过前几次迭代的“热身”之后，负载均衡基本稳定了。从这一点上，我们可以说尽管第二个设备的性能要稍微好一点[⊖]，但是采用的方法透明地实现了几乎最佳的性能。

⊖ 我们以不同的方式验证了性能差异。

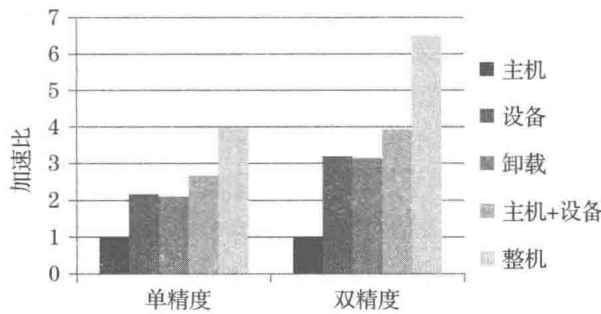


图 11-19 与在两个 Intel Xeon E5-2660 v2 Ivy Bridge 处理器上的性能相比，最新版本的代码在单精度和双精度上的性能加速比（越高越好）

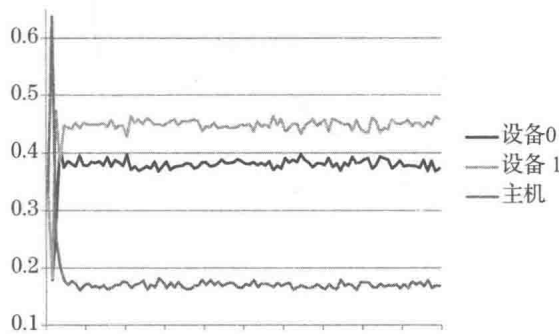


图 11-20 包含开发平台上主机处理器的所有计算设备工作负载分配比例的变化情况

在这个阶段，我们可以认为这份代码已经可以动态调整自己的运行时环境来充分利用所有的可用计算资源。然而，我们依然可以考虑如下改进，以使代码在应对其他可能的情况时更有弹性。

- 数据传输可以独立计时以评估它们对性能的影响。
- 当计算平台中的有些设备与其他设备相比处理速度非常慢（可能由于这些设备的固有性能，也可能由于它们引发的数据传输时间）时，可以考虑完全不使用这些计算设备。

这两个改进都是非常值得探讨的方法，并且应该会被致力于发布最优代码的开发者所考虑。然而，它们都超出了本章的讨论范围，同时，这些优化方法的引入将会使代码可读性降低。

11.6 更多信息

下面是与本章相关的一些额外阅读材料：

- 牛顿运动定律，http://en.wikipedia.org/wiki/Newton%27s_laws。
- 牛顿万有引力定律，http://wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation。
- OpenMP 4.0 标准规范，<http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>。
- 本章及其他章代码下载地址，<http://lotsofcores.com>。

并发内核卸载

Florian Wende^{*}, Michael Klemm[†], Thomas Steinke^{*}, Alexander Reinefeld^{*}

^{*} 德国, 柏林 Zuse 研究所, [†] 德国, Intel

Intel Xeon Phi 产品家族为能够在不同粒度上并行的任务提供了非常强大的计算设备。当需要处理大规模数据时, Intel Xeon Phi 协处理器上所有的可用计算资源都将得到充分利用。

但是, 对于不能充分利用计算资源的小规模工作负载, 协处理器还有吸引力吗?

本章将详细论述当有大量处理小规模数据的任务时, 如何充分利用 Intel Phi 协处理器潜在的计算能力。这是非常重要的, 因为支持多级并行执行流的创新算法数量正在不断增加。当从主机程序的并行实例加载到 Intel Xeon Phi 协处理器上后, 将会为协处理器产生足够的工作负载。

注意 本章优化的最终目标是通过多个内核的并发卸载, 提高小规模工作负载在 Intel Xeon Phi 协处理器上的吞吐量。

12.1 设定上下文

对于很多特定类型的工作负载和生产环境, 多个内核并发卸载是提高 Intel Xeon Phi 协处理器计算吞吐量和计算资源利用率的根本方法。为简单起见, 这里只使用一个协处理器, 可能的应用场景如图 12-1 所示。

- 单个主机进程或线程卸载到一个专用协处理器。这是在今天的 HPC 生产环境中常见的应用场景, 每个计算节点只分配一个用户作业。
 - 每次卸载大规模数据集能够有效利用协处理器的计算资源 (见图 12-1a)。
 - 多个不同计算密度的内核依次卸载, 可能会导致协处理器计算资源的低利用率 (见图 12-1b)。
 - 多个内核并发 (小规模) 卸载, 能够提高设备的利用率 (见图 12-1c)。
- 多个主机进程或者线程卸载到一个共享的协处理器。传统的例子是有复杂工作流程或者并发执行步骤、面向吞吐量的单用户生产环境 (图 12-1d)。
- 卸载到一个远程共享协处理器。如果协处理器虚拟化, 就可在其上运行多个远程节点中的工作负载 (见图 12-1d)。然而, 需要制定有效的调度策略防止资源的过度利用。在并不是每个计算节点都装配协处理器的云计算环境中, 这是常见场景。

本章主要论述在主机使用 MPI 或者 OpenMP 实现并行并且其内核可并发卸载 (如图 12-1c 和图 12-1d 所示) 的应用程序。从小规模负载简单的批处理到存在协同卸载的复杂 (并行) 应用程序, 存在许多这样的实际例子。在介绍这些例子时, 除了描述一些通用方法, 本章还将涉及很多主机和协处理器安装的有用技术。

12.1.1 粒子动力学

粒子动力学 (PD) 模拟的核心是计算 N 个粒子中的粒子 i 受到的其他所有粒子 ($j \neq i$)

的作用力，即 $(F_i = \sum_{j \neq i} F_{ij})$ ，并使用这些作用力，通过运动方程的数值积分来模拟整个系统随时间的发展变化。根据牛顿第三定律 $F_{ij} = -F_{ji}$ ，作用力的对称性可以成倍地减少作用力的计算量。当可以非常容易地串行计算作用力时，并行计算作用力会产生访存限制和同步的困难。这个困难是当计算作用力 F_i 时，会用到来自不同线程的子作用力 F_{ij} 。解决这个问题的一个简单方法是为每个线程分配单独的缓冲区来保存所有的子作用力，然后对所有缓冲区进行合并。然而，内存的消耗会随着线程数目的增多而增加。

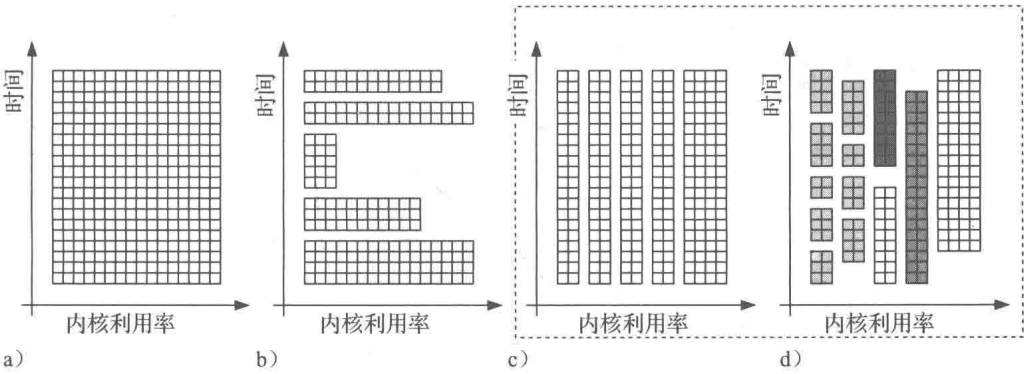


图 12-1 内核卸载到 Intel Xeon Phi 协处理器上的场景——一个主机进程或者线程：a) 利用整个协处理器；b) 多个不同计算密度的内核依次卸载，协处理器计算资源的利用率变化很大；c) 卸载多个并发内核到协处理器上；d) 多个主机进程或线程卸载多个内核到一个共享的协处理器

仅使用一个额外缓冲区，我们就可以做得更好。但是这样会存在显式栅栏同步的开销（见 12.3 节）。图 12-2 显示了使用这个方法进行 PD 模拟的性能。对于所有数据点，在 Intel Xeon Phi 协处理器上卸载一个内核进行处理，用于计算作用力的线程数目从 60 增长到 240。

显然，程序性能的可扩展性并不好。事实上，在 16000 个粒子的情况下，可以接受的可扩展性只能达到 30 个线程。所以，问题是：

我们不再仅卸载一个内核，使 240 个线程处理所有作业。能否通过并发卸载 8 个内核（每个内核使用 30 个线程完成整个系统计算量的 1/8）来提高性能呢？

这个问题的答案是完全可以，但我们需要考虑一系列相关技术。首先介绍这些技术，然后再回到 PD 模拟进行改进。

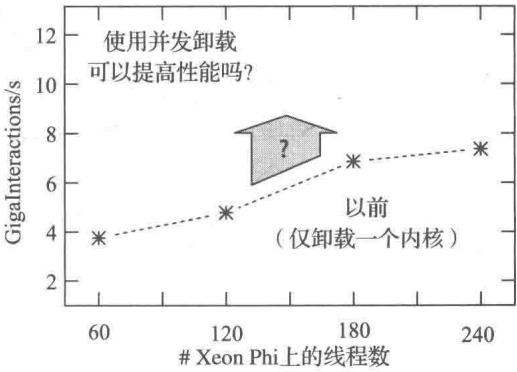


图 12-2 PD 模拟性能：使用牛顿第三定律计算 16 000 个粒子的作用力。性能指标是每秒计算多少粒子间的相互作用，单位为 10^9 ，简称为“GigaInteractions/s”（越大越好）

12.1.2 本章结构

当在 Intel Xeon Phi 协处理器上卸载多个并发内核时，最重要的是设备的划分。在协处理器上分配工作而没有合适设置阶段时，可能会导致性能降低。这是因为线程组可能以一种糟糕的方式相互通信。12.2.1 节将会讲述在并发模式下如何实现线程和计算内核（core）的映射。我们将会使用 Intel MKL（Math kernel Library），通过多个小规模 dgemm 计算来学习

不同关联对性能的影响，并以此来强调线程组织和映射的重要性。

除内核执行之外，主机和协处理器间的数据传输同样会影响程序性能。至少在主机执行的开始和结束阶段是需要数据传输操作的。连续内核卸载操作间的频繁数据传输（常见于迭代算法）会严重降低程序性能。即使当只有一个主机线程或进程将任务卸载给协处理器时，对性能的影响也很大（即使看起来很小）。12.2.2 节将会以并发 MKL dgemm 计算为例，讨论主机和协处理器间的数据传输对性能的影响并会重点讨论使用 MPI 和 OpenMP 进行卸载的重要不同。

最后，12.3 节中，我们将会把并发内核卸载方法应用到 PD 模拟中，并说明处理同样问题时，多个小规模卸载是如何超过一个大规模卸载的。我们在这方面的研究解决了如何协作并发内核卸载的问题。

注意：映射和软件设置

对于协处理器基准测试，本章使用的计算平台配备双插槽 Intel Xeon X5680 处理器（具有 24GB 内存），以及两块安装在 PCIe x16 插槽上的 Intel Xeon Phi 7120P 协处理器。每个协处理器有 61 个物理 CPU 内核，每个内核为四路硬件多线程，并配备 16GB 主存。在实际测试中，我们只使用其中的 60 个内核。这是因为当使用卸载模式时，最好将一个内核用于协处理器上的操作系统。主机运行 CentOS 6.3 linux 操作系统（其内核为 2.6.32-279），Intel MPSS 3.1.4 也安装。代码编译使用 Intel C++Compiler 14.0.3 和 Intel MPI 4.1.1.036。我们使用的编译选项为：`-O3 -xHost -openmp -fno-alias -opt-assumesafe-padding -std=c++11`。

对于处理器基准测试，本章使用的计算平台包含双插槽 Intel Xeon E5 2670 处理器，超线程共有 32 个逻辑 CPU 内核（启用超线程）。该计算平台同时配备 64GB 内存，运行 CentOS 6.3 Linux 操作系统（其内核为 2.6.32-279）。同时也配备了可以和协处理器基准测试系统相媲美的软件栈。

本章代码示例使用 Intel 卸载语言扩展（Language Extension for Offload, LEO）编写。幸运的时，本章代码可以使用新的“目标”指令，在代码改动最少的前提下翻译成 OpenMP 4.0。我们已经这样做了并提供 OpenMP 4.0 版本代码的下载。然而，出于一致性，本章所有的代码段都使用 LEO。LEO 和 OpenMP 4.0 版本的完整源代码可从 <http://lotsofcores.com> 下载。

12.2 协处理器上的并发内核

卸载模式是将不完全并行的主机程序（本机模式并不适用）移植到 Intel Xeon Phi 协处理器上的标准方法。作为与主机相互独立的协处理器，是支持多个卸载程序的同时执行的。然而，如果没有建立合适的线程和内核映射关系，可能会影响多个并发卸载任务的相互通信。

12.2.1 协处理器设备划分和线程关联

Intel Xeon Phi 协处理器可运行 Linux 操作系统。然而，该操作系统不对用户提供服务，而是负责将任务调度到协处理器的 CPU 内核上。为了实现并行，协处理器实现了对 MPI 和 OpenMP 编程模型的原生支持。而在卸载代码部分，OpenMP 是常用方法。和在主机系统上执行 OpenMP 程序一样，可以通过设置环境变量（如 `KMP_AFFINITY` 和 `OMP_PLACES`）获取特定线程的映射。在非并发环境中设置这些环境变量的影响是众所周知的。然而，在

多个主机线程或者进程并发将任务卸载到协处理器上的情况下，却并不是这样。我们使用图 12-3 中的代码说明不同映射机制的影响。特定线程的 CPU 标记位，可通过调用 `sched_getaffinity()` 函数获得（见代码清单）。可通过线程的 CPU 标记位来获得运行这个线程的 CPU 内核。

```
#pragma omp parallel num_threads(m) // m threads on the host
{
    int hostId=omp_get_thread_num();
    #pragma offload target(mic:0)
    {
        #pragma omp parallel num_threads(n) // n threads per offload
        {
            int phiId=omp_get_thread_num();
            cpu_set_t cpuMask;
            sched_getaffinity(0, sizeof(cpu_set_t), &cpuMask);
            printf("thread(hostId=%d, phiId=%d): ", hostId, phiId);
            for(int i=0; i<256; i++)
                if(CPU_ISSET(i, &cpuMask))
                    printf("%d ", i);
            printf("\n");
        }
    }
}
```

图 12-3 确定线程 - 内核映射关系的代码

为了在协处理器上建立环境变量的特定参数（可能与主机不同），我们将环境变量 `MIC_ENV_PREFIC` 的值设定为 `MIC`。在协处理器上触发卸载运行时，以使用分配给 `MIC_XXX` 的参数对 `XXX` 的参数进行重写。

环境变量 `MIC_KMP_AFFINITY` 可以设定为 `compact`、`scatter` 和 `balanced`。环境变量 `MIC_OMP_PLACES` 可以设定为 `threads`、`cores` 和 `sockets`。和预想的一样，我们发现平衡机制没有发挥作用。同时，在 Intel Xeon Phi 协处理器上设定“`socket`”参数是没有意义的，因为只有一个套接字。对于其他映射机制，当使用 OpenMP 和 MPI 卸载任务到协处理器上时，我们将研究在并发模式下线程 - 内核的映射关系。

OpenMP 下的卸载模式

当将环境变量 `MIC_KMP_AFFINITY` 设定为 `scatter` 或者 `compact` 时，协处理器上的每一个线程都只映射到一个逻辑 CPU 内核上。当设定为 `scatter` 时，所有的线程将根据 CPU 内核的逻辑 ID 将线程平均分布到整个系统中。注意，在这种情况下，相邻线程映射到的 CPU 逻辑 ID 跨度可能会大于 1。当设定为 `compact` 时，线程将依次分配到物理 CPU 内核上。当向协处理器并发卸载任务时，卸载部分的并行区域也将并发创建。这样的结果就是无法保证将相同卸载区域的线程映射到连续的（逻辑）内核上。这对于 `scatter` 机制是无关紧要的，因为协处理器的所有物理内核都是一样的。然而，对于 `compact` 机制，相同卸载区域的线程可能会映射到不同的物理内核上，即使它们应该映射到同一个物理内核上。

`MIC_OMP_PLACES=threads` 和 `MIC_KMP_AFFINITY=compact` 的效果是一样的。当 `MIC_OMP_PLACES=cores` 时，线程将会被分配到所有的物理内核上。如果协处理器上开启的线程总数量大于物理内核数量，将会多次分配逻辑内核。不同卸载区域的线程也因此会被分配到相同的物理内核上。这些线程也因此会以一种糟糕的方式彼此相互干扰。

图 12-4 说明了如果 $m = 4$ 个主机线程卸载任务到协处理器上，每个卸载区域开启 4 个线程，不同线程映射机制的影响。在图 12-4 左边部分中，线程 - 内核映射不连续现象特别明显（如“`compact`”模式）。当卸载程序使用缓存优化时，这种情况会导致性能的降低。

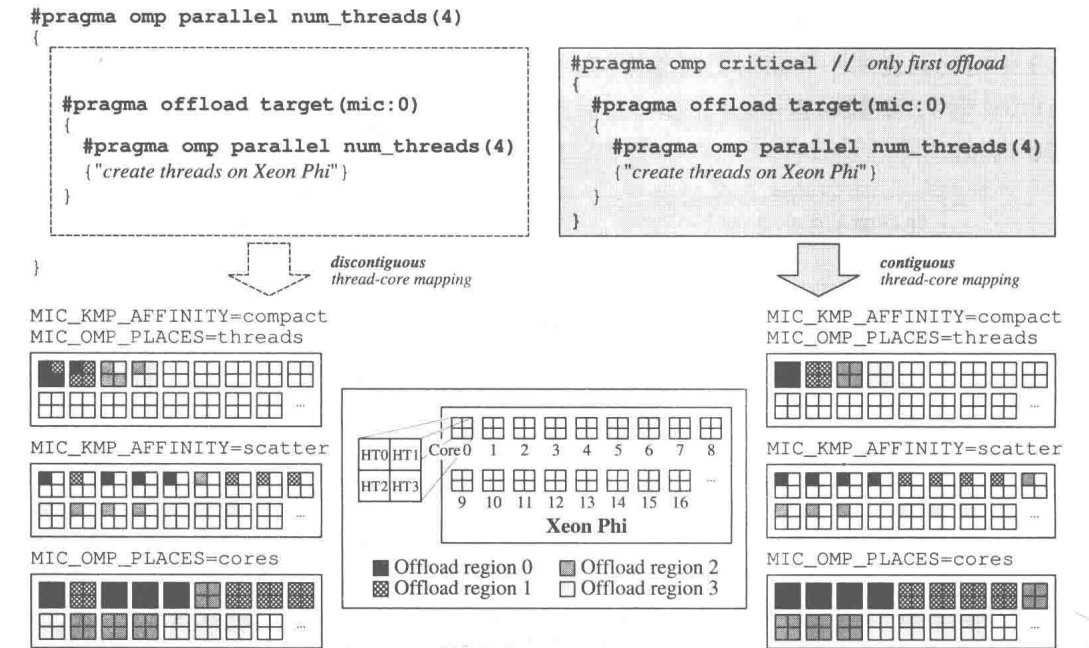


图 12-4 环境变量 MIC_KMP_AFFINITY 和 MIC_OMP_PLACES 设定不同值时对线程 - 内核映射关系的影响。 $m = 4$ 个主机线程将任务并发卸载到协处理器上，每个卸载区域开启 $n = 4$ 个线程。协处理器的物理内核用包含 4 个分块（HT0..3）的正方形表示，每个硬件线程一个分块。不同的填充代表在并发卸载时线程 - 内核的映射关系。左边的映射是非连续的；相同卸载区域的线程没有映射到连续的（逻辑）内核上（见“compact”机制）。右边的映射修正了这个行为，从程序开始处卸载到协处理器上是相互排斥的

消除并发卸载情况下线程 - 内核映射不连续现象的一个简单解决（修正）方法是：从程序开始使卸载操作是互斥的，并且在真正的卸载计算执行之前，在卸载区域内部开启所有需要的线程。如图 12-4 右边部分所示，我们需要在图 12-3 所示代码中的第一次卸载区域设置一个 Critical 区。

然而，即使如此，执行两次程序并不保证相同的线程 - 内核映射。为了解决这个问题，如图 12-5 所示，可以通过使用 Linux 调度接口（例如，`sched_setaffinity()`），显式将线程分配到内核上。

```
#pragma offload target(mic:0)
{
    #pragma omp parallel num_threads(4) // four threads on Xeon Phi
    {
        int threadId=omp_get_thread_num();
        cpu_set_t cpuMask;
        CPU_ZERO(&cpuMask); // clear cpuset
        CPU_SET(1+2*4+threadId,&cpuMask); // use 3rd physical core
        sched_setaffinity(0, sizeof(cpu_set_t), &cpuMask);
    }
}
```

图 12-5 调用 Linux 调度接口完成线程 - 内核的显式映射。在这个例子中，每个卸载区域开启 $n = 4$ 个线程，显式地将所有线程映射到第三个物理内核上

注意 当在多线程应用程序中进行并发卸载时，使卸载操作互斥进行，并且在真正的卸载计算执行之前开启协处理器侧的线程是非常有意义的。在这种方式下，卸载区域的线程关联将会和 MIC_KMP_AFFINITY 以及 MIC_OMP_PLACES 设置的参数一致。

MPI 下的卸载模式

MPI 主机程序使用不同 MPI 进程进行并发卸载时，会产生困难。因为，在理论上，每个进程都假设会以互斥的方式同其他进程使用协处理器。

设置诸如 MIC_KMP_AFFINITY 或者 MIC_OMP_PLACES 等环境变量仅会影响单个进程，但不能解决协处理器的划分和线程映射问题。特别是，每个进程的卸载运行时和它们的卸载操作并不协调。因此，可能会将协处理器上的线程映射到同一个执行单元，从而导致协处理器内核的过度使用。

针对这个问题一个显而易见的方法是实现一个显式的线程划分（thread pinning）方案，例如，使用 sched_setaffinity() 函数，结合一些逻辑，根据进程数划分设备。然而，通过设置环境变量 MIC_KMP_PLACE_THREADS=XXc,YYt,ZZO 也能实现相同功能。该环境变量允许将协处理器划分成多个块。例如，分块（XXc,YYt,ZZO）包含一组从 ZZ 开始的 XX 个连续的协处理器内核。YY ∈ {1,2,3,4} 为每个内核可用的硬件线程数目。环境变量 MIC_KMP_AFFINITY 和 MIC_OMP_PLACES 定义了当卸载操作执行时，块内线程 - 内核的映射。

例如，如果使用 $m = 2$ 个 MPI 进程将计算卸载到一块 Intel Xeon 7XXX Phi 协处理器上，每次卸载开启 $n = 60$ 个线程。我们可以这么做：

```
$> mpirun -genv MIC_OMP_NUM_THREADS=60
-np 1 -env MIC_KMP_PLACE_THREADS=30c,2t,00 ./prog.x :
-np 1 -env MIC_KMP_PLACE_THREADS=30c,2t,300 ./prog.x
```

注意 在 MPI 应用程序中，要建立卸载区域内一个特殊的线程划分机制，需要在协处理器上定义显式的线程 - 内核映射，或者需要设备划分和线程映射，例如通过 MIC_KMP_PLACE_THREADS 和 MIC_KMP_AFFINITY 或 MIC_OMP_PLACES 两个环境变量。

案例研究：Intel MKL dgemm 并发卸载

BLAS（basic linear algebra subprograms，基本线性代数子程序）对许多科学计算代码非常重要。对于通过线程或者进程的执行路径调用大量矩阵 - 矩阵乘法计算的并行应用程序，如果将这部分工作移植到协处理器上以更快执行，很有可能提升整个程序的性能。

现在我们考虑如何使用多个并发 OpenMP 线程和 MPI 进程将 MKL dgemm 操作卸载到协处理器上执行。dgemm 是用来评估不同线程 - 内核映射机制对整个程序性能影响的一个非常合适的例子。这主要有两方面的原因：第一，MKL dgemm 是经过缓存优化的，所以不好的线程 - 内核映射应该会对程序的整体性能产生消极影响。第二，这个案例允许我们实际研究线程核 - 内核间的映射，而不会受到共享协处理器主存带宽的所有并发卸载操作的影响。

图 12-6 是基准测试程序的核心代码。主机和协处理器间的数据传输将会在 12.2.2 节中进行讨论。现在，主要讨论卸载计算。

注意：并发内核的计时

为了在协处理器上评估多个并发计算的性能：首先我们取得每个线程或者进程的每一个并发卸载（有多个）的开始和结束时间，然后确定一个时间段（并发窗口），在这个并发窗口内，所有的进程或者线程都有重叠的卸载操作。并发窗口可通过所有进程或者线程的第一次卸载的最晚开始时间和最后一个卸载操作的最早结束时间获得。在这个时间段内，我们假设 100% 的并发。通过计算落在并发窗口的卸载操作的数量，我们可以获得性能指标。


```

..
double start[15][numOffloads], stop[15][numOffloads];
#pragma omp parallel num_threads(15)
{
    double *a=_mm_malloc(N*N*sizeof(double), 64), *b=., *c=.;
    // #pragma omp critical // our "fix"
    // {
    // initialize a[], b[] and create persistent buffers on Xeon Phi for a[], b[] and c[]
    #pragma offload target(mic:0)\
    in(a:length(N*N) align(64) alloc_if(1) free_if(0))\
    in(b:length(N*N) align(64) alloc_if(1) free_if(0))\
    nocopy(c:length(N*N) align(64) alloc_if(1) free_if(0))
    {
        mkl_set_num_threads(16); // use 16 threads on Xeon Phi
        #pragma omp parallel num_threads(16)
        {
            ; // just create threads! MKL uses OpenMP.
        }
    }
    // }
    #pragma omp barrier
    for(int i=0; i<numOffloads; i++){ // now start the offload computations
        start[omp_get_thread_num()][i]=get_time_stamp();
        #pragma offload target(mic:0)\
        in(a:length(0) alloc_if(0) free_if(0))\
        in(b:length(0) alloc_if(0) free_if(0))\
        out(c:length(0) alloc_if(0) free_if(0))
        {
            cblas_dgemm(..); // matrix-matrix multiplication: c=a*b
        }
        stop[omp_get_thread_num()][i]=get_time_stamp();
    }
    // release memory a[], b[] and c[] on host and Xeon Phi
}
// determine concurrent performance: extract concurrency window from start[][], stop[][]
..

```

图 12-6 MKL 中 dgemm 基准测试的核心代码：矩阵规模为 $N \times N$ ，使用 $m = 15$ 个主机线程，每次卸载开启 $n = 16$ 个协处理器线程，这样总共开启了 240 个线程，与 Intel Xeon Phi 协处理器的逻辑核数量一致。协处理器上缓存的分配由 `alloc_if` 和 `free_if` 管理。在协处理器和主机间没有数据传输

在 dgemm 这个例子中，性能指标通过用并发窗口中卸载操作的数量乘以每次矩阵-矩阵乘法计算的浮点数操作数量，然后除以并发窗口长度获得。通过这种方法，我们可以获得每次执行的 FLOPS (floating point operations per second, 每秒执行的浮点操作次数)。在所有的例子中，我们的方法都会低估程序性能，但不会高估。这是非常重要的，否则这里的结果就过于乐观了。

图 12-7 显示了并行执行 $m = 15$ 个卸载操作，每个使用 $n = 16$ 个 OpenMP 线程时，在 Intel Xeon Phi 协处理器上的浮点性能。测试规模为 2048×2048 的矩阵在不同线程-内核映射机制下的性能。每个主机线程或者进程卸载 100 个 dgemm 计算到协处理器上。我们比较了分别使用 MPI 和 OpenMP 执行卸载操作的性能。当使用 MPI 时，我们使用 `MIC_KMP_PLACE_THREADS` 环境变量来实现需要的设备划分。

图 12-7a 说明了通过使用 `sched_setaffinity()` 实现协处理器上显式线程划分与使用环境变量（环境变量没有使用定义的值）时的性能差距。显式线程划分类似于“compact”机制。在这种情况下，执行相同矩阵-矩阵乘的线程将共享相同的物理内核的 L1 缓存。因此，当环境变量没有设置为定义的值时，非连续的线程-内核映射的缺点就非常明显了。当使用它时，“compact”机制接近于显式线程划分，所以我们只显示了“compact”机制的性能图。另外，可以看出“scatter”机制下，真实的线程-内核映射并不重要，因为协处理器上的物理内核都是一样的。

图 12-7b (MPI) 只包含了 “scatter” 和 “compact” 两种机制下的性能情况。这两种机制都是在恰当划分协处理器后，通过设置 MIC_KMP_AFFINITY=[scatter|compact] 建立的。可以看出，性能情况和使用主机线程类似。然而，也可以看出，MPI 环境的建立引入了开销从而引起了性能的小幅下降。

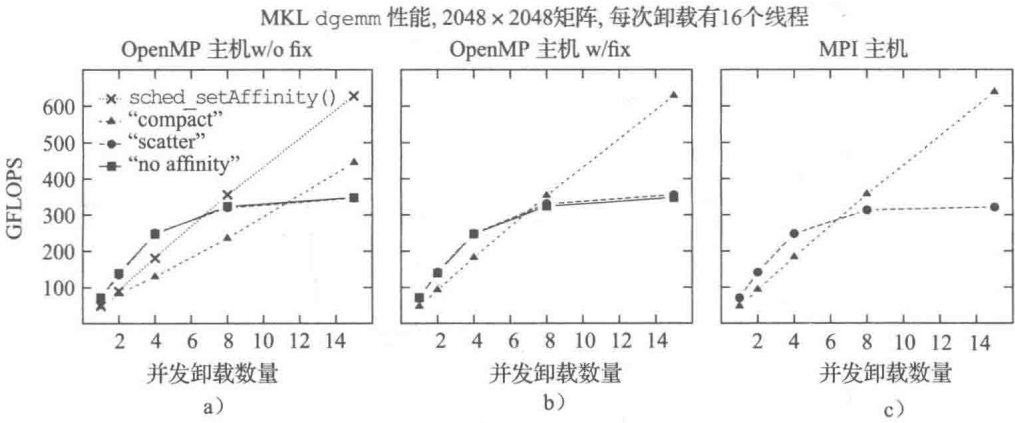


图 12-7 MKLdgemm 在 Intel Xeon Phi 7120P 协处理器上的性能。总共有 15 个并发卸载操作，每个卸载操作使用了 16 个 OpenMP 线程。性能图显示了不同线程 - 内核映射对整体性能的影响。图 b 和图 c 没有包括通过 sched_setaffinity() 设置显式线程划分时的性能。因为它们和 “compact” 机制一样。当在主机执行 MPI 时，只显示了 “compact” 和 “scatter” 两种机制

协处理器上的持久性线程组和线程关联

LEO 和 OpenMP 4.0 都使用 COI (Coprocessor Offload Infrastructure) API 来创建协处理器端的进程和线程 (线程组)、内存缓冲区以及协处理器和主机间的通信通道。尤其是对于每个卸载任务到协处理器的主机进程，都会在协处理器上有一个对应的进程。在进程的上下文中，至少存在和通过执行路径卸载编译指令的主机线程数目一样的线程。主机线程和对应的协处理器线程通过 COIPipeline 相连 (见 Newburn et al., 2013)。COIPipeline 为协处理器上的内核调用实现了一个 FIFO 命令队列，并会进行高达几千字节的数据传输 (更多的数据传输使用 COIBuffer 实现)，这通常会有通过 COIPipeline 调用内核触发。这其中重要的一点是：通过 COIPipeline 构建主机与协处理器线程和线程间的连接，维持了在协处理器上 OpenMP 并行区域的线程以及它们对应的关联。

注意 除非并行区域的形状改变，否则连续的 (并发) 卸载操作会继承之前卸载操作的线程配置。因此，在应用程序开始时，使用的特定线程划分机制在整个执行过程中都会有效。

12.2.2 并发数据传输

当将计算卸载到协处理器后，至少在计算的开始和结束都会发生一次主机和协处理器间的数据传输。数据传输也时常会在中间发生，比如相对独立的连续卸载操作。尽管在并发卸载的情况下，每个主机线程都有一个独立的 COIPipeline 来进行内核的调用，但在目前的 COI 实现中，同一个进程上下文的所有数据传输会使用共享的数据通道。所以，当在主机端使用多个 OpenMP 线程时，数据传输会受到共享 COI 资源隐式串行和竞争的影响，从而也有可能导致内核串行执行。

在主机端使用 MPI 进行数据传输时能够使用多条数据通道。对于在主机和协处理器间会有频繁数据传输的应用程序而言，在主机上使用 MPI 代替线程应该会获得性能提升。然而，这仅对小消息有效。对于大消息，特别是一次数据传输就可以充分利用 PCIe 的传输带宽时，并发的数据传输实际上串行化了。

图 12-8 说明了当主机使用 OpenMP 或者 MPI 时，针对不同大小的数据块（消息）和不同并发数据传输数量的数据传输时间 t 。所有的数据传输时间 t 都是通过在前面介绍 MKL 性能评估时使用的“并发窗口”方法测得的。

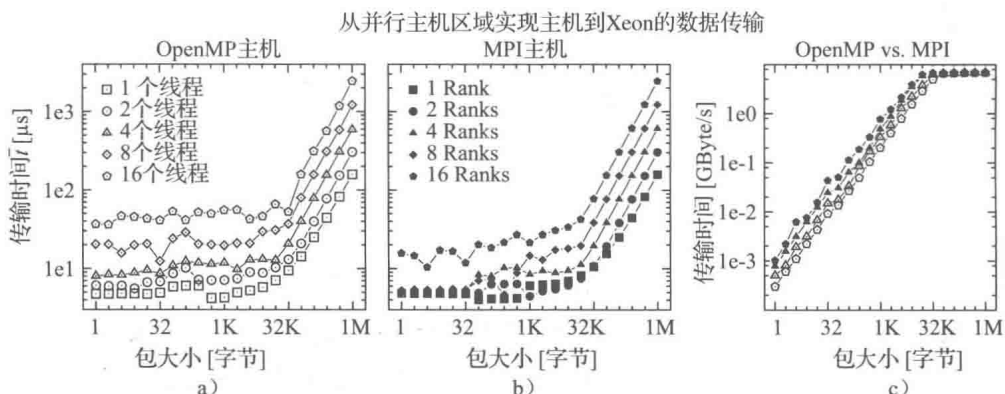


图 12-8 从主机到协处理器传输不同大小的数据块时的传输时间 \bar{t} 和输出速率。该图说明了当使用 OpenMP 或者 MPI 时，增加并发数据传输次数对性能的影响。我们使用双插槽 Intel Xeon X5680 处理器和 Intel Xeon Phi 7120P 协处理器（插在 PCIe × 16 插槽中）

图 12-8 中的性能点是从 25 次独立的程序运行中获取的最短数据传输时间。对于每次执行，并发窗口（也就是时间 t ）都是由每个线程或者进程执行了 5000 次数据传输（针对不同大小的数据包）后测得的。

在图 12-8a 中，当数据块不超过 16KB 时，数据传输时间几乎和主机线程数目成正比。因为每个进程只有一个数据通道，所以尽管当数据包小于 16KB 时并不能占满 PCIe 的总线带宽，但在任意时刻只有一个这样的数据包在执行。当使用 MPI 时，对于每个 MPI 进程都会有一个数据通道存在。对于不超过 64 字节大小的数据包，数据传输时间几乎相等（独立于使用的 MPI 进程数量）。当数据块大小等于 64 字节时，SCIF (Symmetric Communications InterFace, 对称通信接口) 实现将从非 DMA 转变为 DMA（小数据量的传输由 CPU 自己管理）。SCIF 为 COI 使用的通信后端。当数据包增大到 16KB 时，一个数据包就足以占满 PCIe 的带宽，所以使用 OpenMP 和 MPI 的数据传输时间相等。

注意 如果主机和协处理器间频繁进行小量的数据传输，同时内核的执行时间和数据的数据传输时间在同一个数量级上，使用 MPI 进行并发卸载可能会产生最佳性能。

案例研究：考虑数据传输的并发 MKL dgemm 卸载

到目前为止，我们还没有研究在内核执行间进行并发数据传输对性能的影响。对此，我们扩展了图 12-6 展示的 MKL dgemm 基准测试，即在卸载计算前后添加了额外的数据传输操作，如图 12-9 所示。矩阵元素的传输分别由 x_A 、 x_B 和 x_C 控制。

当 $x_A = 1.0$ ， $x_B = 0.0$ ， $x_C = 1.0$ 时，一个常数矩阵 b （如在量子力学中操作数的矩阵表示），反复作用于矩阵 b 后，其计算结果矩阵 c 返回主机端。当 $x_A = 1.0$ ， $x_B = 1.0$ ， $x_C = 1.0$ 时，所有的问题就是协处理器和主机间的数据传输了。图 12-10 说明了当分别使用 OpenMP

和 MPI 进行卸载操作时的整体性能。在协处理上使用“compact”划分机制。

```
.. // See Figure 12.6
#pragma omp barrier // now start the offload computations
for (int i=0; i<numOffloads; i++) {
    start[omp_get_thread_num()][i]=get_time_stamp();
    #pragma offload target(mic:0)\
    in(a:length(xA*N*N) alloc_if(0) free_if(0))\ // ← data transfer
    in(b:length(xB*N*N) alloc_if(0) free_if(0))\ // ← data transfer
    out(c:length(xC*N*N) alloc_if(0) free_if(0))\ // ← data transfer
    {
        cblas_dgemm(..); // matrix-matrix multiplication: c=a*b
    }
    stop[omp_get_thread_num()][i]=get_time_stamp();
}
..
```

图 12-9 对图 12-6 代码的扩展。现在，每个卸载计算被主机和协处理器间的数据传输包围。进行数据传输的数组大小分别由 xA、xB 和 xC ∈ [0,1] 决定

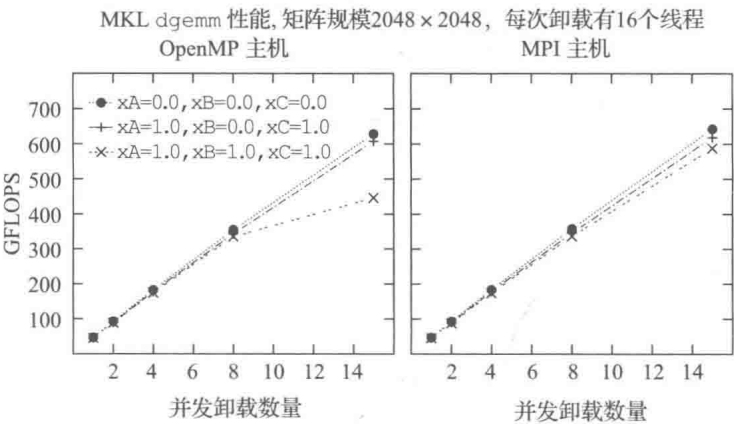


图 12-10 MKL gdemm 在 Intel Xeon Phi 7120P 协处理器上的性能（协处理器在 PCIe × 16 插槽中）。总共有 15 个并发卸载操作，每个卸载操作在协处理器上有 16 个 OpenMP 线程。该图说明了在连续卸载操作间进行数据传输时对整体性能的影响

当在主机端使用 OpenMP 时，因为每个进程只有一个数据传输通道，所以数据传输就会串行化。相反，如果在主机端运行多个 MPI 进程，每个进程都会有独立的数据传输通道。然而，当矩阵的规模超过 32MB 时，一个线程的数据传输就可以占满整个 PCIe 带宽。数据传输的串行化也因此对整体性能没有影响。更进一步讲，这和使用 MPI 几乎没有任何差别，这是因为进行数据传输的多个数据通道共享 PCIe 带宽，所以使用 MPI 进行并发数据传输相对于串行传输没有性能提升。

然而，当在主机上使用 OpenMP 时，随着并发卸载操作数量的增加，我们观察到数据传输导致了一个明显的性能损失。对于 2048 × 2048 的矩阵规模，计算时间（对于 16 个线程为 380ms，性能为 45GFLOPS）和数据传输时间（对于三个矩阵为 15ms，性能为 6.8GB/s）之间相差 25 倍。这种情况下，应该可以用计算隐藏总共 25 个线程或者进程的数据传输。当使用 MPI 时，这基本上是可以实现的。

注意 多线程并发卸载操作需要面临 COI 内部数据结构的竞争，而当使用 MPI 时，这个数据结构会复制多份。因为对于这些内部数据结构的竞争没有更好的解决办法，所以在和 Intel Xeon Phi 协处理器进行数据交换时，或者只使用中等数量的并发卸载线程，或者转变为 MPI 或者 MPI+OpenMP 混合模式，这样应该是最最好的。

12.3 在 PD 中使用并发内核卸载进行作用力计算

现在, 我们回到 PD 例子上来, 这个例子一开始就是为了说明在协处理器上并发卸载内核而设立的。请记住一点: 我们本来计划利用作用力的对称性, 但发现随线程数量增多关于作用力计算的实现的可扩展性并不好。接下来, 我们将描述评估作用力的并行算法, 并且对使用 C++ 和 Intel LEO 实现的部分进行详细描述。

12.3.1 使用牛顿第三定律并行评估作用力

根据牛顿第三定律: $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$, 其中, \mathbf{F}_{ij} 是粒子 i 对粒子 j 产生的作用力。利用这种对称性, 当对于 N 个粒子评估每个粒子所受的总作用力 ($\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij}$) 时, 可以减少两倍的计算量: 我们需要计算 ($\mathbf{F}_i = \sum_{j < i} \mathbf{F}_{ij}$), 然后加上 $\mathbf{F}_{ji} = -\mathbf{F}_{ij}$ 对总作用力的贡献。为简单起见, 我们直接评估作用力 \mathbf{F}_i , 这将导致一个复杂度为 $O(N^2)$ 的算法。此外, 我们认为粒子间的作用力 \mathbf{F}_{ij} 只依赖于粒子间的空间距离 $x_{ij} = |\mathbf{x}_i - \mathbf{x}_j|$ 以及它们的电量 q_i 和 q_j 。如:

$$\mathbf{F}_{ij} = \left(a \frac{b^6}{x_{ij}^8} \left(\frac{2b^6}{x_{ij}^6} - 1 \right) + \frac{q_i q_j}{x_{ij}^3} \right) (\mathbf{x}_i - \mathbf{x}_j)$$

通过兰纳-琼斯势和库仑势获得 (见 Thijssen, 2013)。

导致 $O(N^2)$ 算法在协处理器上并行实现困难的原因是什么? 第一, 因为协处理器上的“线程要求” (见 Jeffers and Reinders, 2013), 即每个物理内核至少要实例化两个硬件线程; 第二, 利用对称性计算作用力必须是线程安全的, 这对数百个线程来说是一件非常不容易的事情。

线程安全要求是因为作用力 \mathbf{F}_i 会并行接收来自于不同线程的作用力分量 \mathbf{F}_{ij} 。解决这个问题一个显而易见的方法是对 P 个线程中的每个线程都分配一个额外的缓冲区来存储作用力分量 $\mathbf{F}_{ji} = -\mathbf{F}_{ij}$, 随后累计这些缓冲区到 \mathbf{F}_i 中。然而, 内存消耗会随线程数目的增多以复杂度 $O(P \cdot N)$ 的速度“剧增”。一个空间复杂度为 $O(N)$ 的更好方法为:

并行算法 (无并发): (0a) 创建两个长度为 N 的缓冲区: \mathbf{F} 和 \mathbf{F}^* , 并设置 $\mathbf{F} \equiv 0$, $\mathbf{F}^* \equiv 0$ 。(0b) 使用 P 个线程进行作用力 \mathbf{F}_i 的计算。现在执行如下迭代, 设 k 的初始值为 0:

(1) 线程 $p \in [0, P-1]$ 执行如下计算:

```
for i in "my i-values" do
  for j = mod(p + k, P) to i - 1 increment by P do
    f = "compute  $\mathbf{F}_{ij}$ ",  $\mathbf{F}[i] = \mathbf{F}[i] + \mathbf{f}$ ,  $\mathbf{F}^*[j] = \mathbf{F}^*[j] - \mathbf{f}$ 
  end for
end for
```

(2) 设置 $k = k+1$, 如果 k 等于 P , 则合并缓冲区, 即: $\mathbf{F} = \mathbf{F} + \mathbf{F}^*$ 并退出; 否则, 同步线程并继续执行步骤 1)。

步骤 (1) 会执行 P 次。在每次迭代中, 线程通过计算作用力分量 \mathbf{F}_{ij} 访问缓冲区 \mathbf{F}^* 的第 j 个元素, j 的初始值为 0, 间隔为 P 。所有线程的操作都是唯一的——见第二层循环中的 $\text{mod}(p + k, P)$ 。在连续迭代间添加线程同步, 空间复杂度为 $O(N)$ 的线程安全就实现了。

图 12-2 说明了当在 Intel Xeon Phi 协处理器上开启 240 个 OpenMP 线程计算 16000 个粒子时, 该算法的性能 (使用了基于 SIMD 优化的代码库)。该算法的可扩展性只支持至多 30 个线程。

并行算法 (并发): 和上一个算法只使用一个大小为 P 的线程组不同, 我们使用 m 个大小为 $P^\dagger = P/m$ 的线程组。现在这些线程都有一个组标识符: $g^\dagger \in [0, m-1]$ 、一个组内标识符 $[0, P^\dagger-1]$ 、一个全局线程标识符: $g^\dagger m + p^\dagger$ 。全局线程标识符用于在所有 P 个线程中设置 i 的值。

通过 $(0a)^\dagger$ 每个线程组分配两个缓冲区: F^\dagger 和 $F^{*\dagger}$; $(0b)^\dagger$ 将它们初始化为 0, 我们在线程组内部使用非并发算法 (k 初始化为 0):

(1) † 在步骤 (1) 中进行如下替换: $p \rightarrow p^\dagger, P \rightarrow P^\dagger, F \rightarrow F^\dagger, F^* \rightarrow F^{*\dagger}$ 。

(2) † 设置 $k = k+1$: 如果 k 等于 P^\dagger , 则合并缓冲区, 即 $F^\dagger = F^\dagger + F^{*\dagger}$, 退出; 否则, 在线程组内进行线程同步, 然后继续执行步骤 (1) † 。

在并发情况下的迭代次数为 $P^\dagger < P$, 在每个线程组内进行同步的线程数目也减少到 P^\dagger 。为了得到作用力 F_i 的最终值, 有必要进行步骤 (3) 以累加所有 m 个线程组的作用力。

图 12-11 说明了主机端使用 8 个线程进行并发卸载操作时的并行算法。注意, 步骤 $(0a)^\dagger$ 没有在图中显示。步骤 $(0b)^\dagger$ 、 $(1)^\dagger$ 和 $(2)^\dagger$ 在协处理器上执行, 每次卸载操作, 协处理器开启了 30 个线程。为了区分缓冲区 F^\dagger 和 $F^{*\dagger}$, 我们引入了下标 “ $(g^\dagger + 1)$ ”。对于线程和卸载操作的相关元素, 我们使用 OpenMP 和 LEO 编译制导指令。在步骤 $(3)^\dagger$ 之前, 所有主机线程都进行了同步, 并将任务卸载到协处理器上进行作用力的计算。

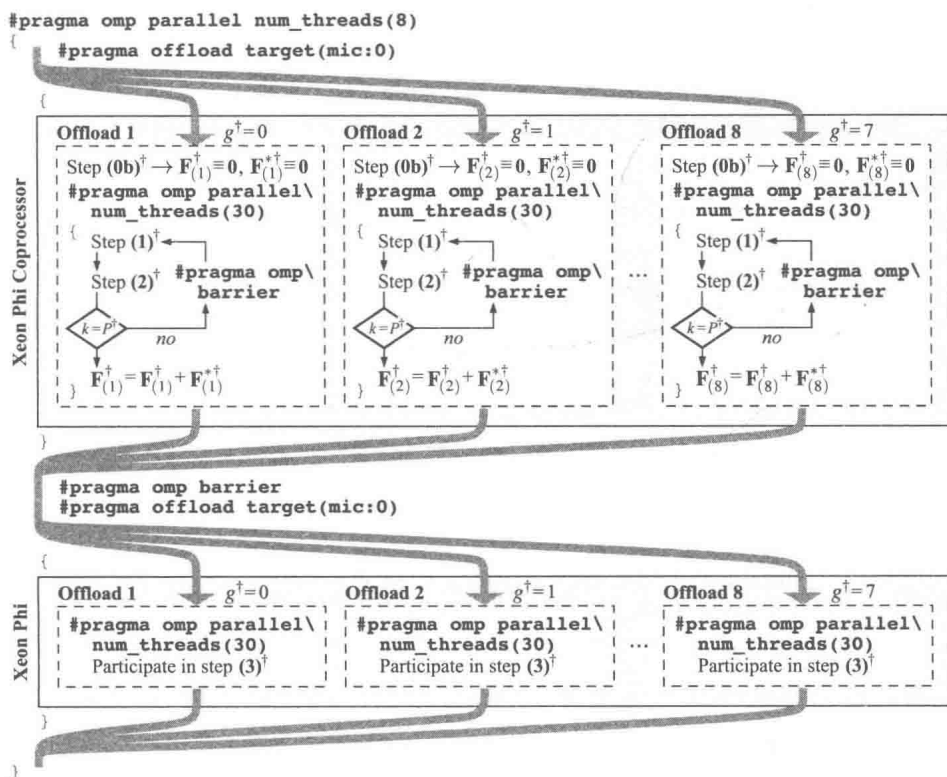


图 12-11 将作用力计算和累加操作卸载到 Intel Xeon Phi 协处理器上的结构图: 8 个主机线程分别将 1/8 的任务卸载到协处理器上, 每次卸载操作开启 30 个线程来处理子任务

12.3.2 实现作用力并发计算

现在我们开始进行使用并发卸载的作用力并行计算算法的实现 (全部源代码, 包括使用循环展开和分块等 SIMD 优化的代码, 可以从 <http://lotsofcores.com> 下载)。在真正进入实

现细节之前，我们需要明确一个在并发卸载间共享数据的方法，这可能在计算作用力时对步骤 (3)[†] 有好处。

注意：使用 OpenMP 时，在并发卸载间共享协处理器数据

同一个主机进程的卸载计算也共享相对应的协处理器端进程。所以，任何主机线程在执行过程中分配的内存都可以被属于同一个进程上下文的其他主机线程使用，只要相关主机指针在这些线程之间共享。

下面说明了当有 8 个并发卸载操作时，使用共享内存以数组索引值填充指针 ptr 指向数组的元素。这种方法之所以能够正确运行，是因为当卸载任务到 Intel Xeon Phi 协处理器上时，Intel 的 LEO 和 OpenMP 4.0 都使用了 COI，而 COI 将会以上文描述的方式工作，并且不太可能在这一方面进行改变。

```
int *ptr= mm_malloc(512*sizeof(double),64);
#pragma offload transfer target(mic:0)\
in(ptr:length(512) align(64) alloc_if(1) free_if(0))

#pragma omp parallel num_threads(8) // 8 threads on host
{
    int id=omp_get_thread_num();
    #pragma offload target(mic:0)\
    in(ptr:length(0) alloc_if(0) free_if(0)) // ← shared memory
    {
        for(int i=id; i<512; i+=8)
            ptr[i]=i; // concurrent offloads can use the shared memory pointed to by ptr
    }
}

#pragma offload transfer target(mic:0)\
out(ptr:length(512) alloc_if(0) free_if(1))
// the content of the memory pointed to by ptr now is: 0 1 2 3 .. 511
```

作用力计算是 PD 模拟的核心部分，其基本功能使用 C++ 语言在类 pd 中实现。图 12-12 展示了该类的声明。常量 CONCURRENT_OFFLOADS 和 THREADS_PER_OFFLOAD 分别定义了并发卸载操作的数目 m 以及每次卸载操作使用的线程数量 p^{\dagger} 。

```
#define ALLOC align(64) alloc_if(1) free_if(0)
#define REUSE alloc_if(0) free_if(0)
#define FREE alloc_if(0) free_if(1)

#define CONCURRENT_OFFLOADS (8) // corresponds to m
#define THREADS_PER_OFFLOAD (30) // corresponds to P†

// Class pd: provides the basic functionality for a PD simulation.
class pd{
public:
    pd(double *position,double *charge,...,int numParticles);
    ~pd();
    void computeForce();
    .. // see source code downloadable from http://lotsofcores.com.
private:
    int numParticles,deviceId=0;
    double *position,*charge,*force_i,*force_j[CONCURRENT_OFFLOADS];
    ..
};
```

图 12-12 C++ 中 pd 类的声明，只列出了相关部分

构造函数 (见图 12-13)：在主机和协处理器上分配描述粒子 position 和 charge 的内存以及用于作用力计算的缓冲区 force_i 和 force_j[0..(CONCURRENT_OFFLOADS-1)] (对应于每个线程组的缓冲区： F^{\dagger} 和 $F^{*\dagger}$)。在卸载区域用到的属性不得不存放在构造函数的栈中。如果不这样做，例如，对于属性 position，编译器将会把每个出现的 position 都编译进 this->position 中，这在协处理器上是无效的，因为这样的 pd 对象根本就不存在。这样，当在协处理器上访问这个指针时，将会导致段错误。为解决这个问题，创建了

该属性的一个本地副本。对于本地副本，卸载运行时将会在协处理器上创建对应的对象。

```
// Constructor: particle positions have format (x1,...,xN), (y1,...,yN), (z1,...,zN). Factor "3."
pd::pd(double *position, double *charge, ..., int numParticles) {
    int sizeBytes=numParticles*sizeof(double);
    this->numParticles=numParticles;
    this->position=_mm_malloc(3*sizeBytes, 64);
    this->charge=_mm_malloc(sizeBytes, 64);
    memcpy(this->position, position, 3*sizeBytes);
    memcpy(this->charge, charge, sizeBytes);
    // create buffers force_i and force_j[]
    this->force_i=_mm_malloc(3*sizeBytes*CONCURRENT_OFFLOADS, 64);
    for(int c=0; c<CONCURRENT_OFFLOADS; c++)
        this->force_j[c]=_mm_malloc(3*sizeBytes, 64);
    // allocate and set up position, charge, force_i and force_j[] on the coprocessor
    double *pos=this->position, *chg=this->charge, *f_i=this->force_i;
    #pragma offload_transfer target(mic:deviceId) \
    in(pos:length(3*numParticles) ALLOC) \
    in(chg:length(numParticles) ALLOC) \
    nocopy(f_i:length(3*numParticles*CONCURRENT_OFFLOADS) ALLOC)
    #pragma omp parallel num_threads(CONCURRENT_OFFLOADS)
    {
        double *f_j=this->force_j[omp_get_thread_num()];
        #pragma omp critical
        {
            #pragma offload target(mic:deviceId) \
            nocopy(f_j:length(3*numParticles) ALLOC)
            {
                #pragma omp parallel num_threads(THREADS_PER_OFFLOAD)
                {
                    ; // just create threads
                }
            }
        }
    }
}
```

图 12-13 pd 类的构造函数。属性 force_i 和 force_j[] 需要在构造函数栈中转化为本地变量。为什么这是必需的呢？force_j 实际上映射 this->force_i，但这个变量在协处理器上是无效的，因为这样的 pd 对象本就不存在。在协处理器上访问 this->force_i 将会导致段错误

force_i 指向一块大小为 $\text{CONCURRENT_OFFLOADS} \cdot N$ 的内存，存储了所有并发卸载操作的作用力计算： $F_i = \sum_{j \neq i} F_{ij}$ （每次卸载操作大小为 N 的相互不相交的子数组）。相反，force_j[] 包含了 "CONCURRENT_OFFLOADS" 个指针，每个指针指向一块大小为 " N " 的内存区域。force_j[] 是分布计算 $F_{ji} = -F_{ij}$ 时每次卸载操作的私有变量；而 force_i 是在作用力归约步骤中用于所有并发卸载操作的共享内存。所以我们希望它是连续的。

接下来，依据在 12.2.1 节的描述，在协处理器上创建了所有 OpenMP 线程。设置 MIC_KMP_AFFINITY=compact，以使在相同卸载区域的所有线程都映射到协处理器上连续的逻辑内核中。

析构函数（见图 12-14）：指针类型的“属性”都存放在析构函数的栈上，相对应的内存区域按照先协处理器后主机的顺序释放。

方法 computeForce()（见图 12-15）：主机端的作用力计算。再一次强调，所有相关属性都存放在函数的栈上以便在卸载区域访问它们。在这个函数中，创建了 CONCURRENT_OFFLOADS 个主机线程，每次卸载操作首先将作用力计算函数 __computeForce() 卸载到协处理器上，然后是作用力累加函数 __accumulateForce()。两次卸载操作在主机端被栅栏同步操作分开。


```

// Destructor: release coprocessor and host memory
pd::~pd() {
    double *pos=this->position,*chg=this->charge,*f_i=this->force_i;
    #pragma offload target(mic:deviceId)\
    in(pos,chg,f_i:length(0) FREE)
    {
        ; // just release memory
    }
    _mm_free(this->position);
    _mm_free(this->charge);
    _mm_free(this->force_i);
    for(int c=0;c<CONCURRENT_OFFLOADS;c++){
        double *f_j=this->force_j[c];
        #pragma offload target(mic:deviceId)\
        in(f_j:length(0) FREE)
        {
            ; // just release memory
        }
        _mm_free(this->force_j[c]);
    }
}

```

图 12-14 pd 类的析构函数

```

// Concurrent force computation: host part of the force computation.
void pd::computeForce() {
    int n=this->numParticles;
    double *pos=this->position,*chg=this->charge,*f_i=this->force_i;
    #pragma omp parallel num_threads(CONCURRENT_OFFLOADS)
    {
        int offloadId=omp_get_thread_num();
        double *f_j=this->force_j[offloadId];
        #pragma offload target(mic:deviceId)\
        in(pos,chg,f_i,f_j:length(0) REUSE)
        {
            __computeForce(pos,chg,&f_i[3*n*offloadId],f_j,n,offloadId);
        }
        #pragma omp barrier
        #pragma offload target(mic:deviceId)\
        in(f_i:length(0) REUSE)
        {
            __accumulateForce(f_i,n,offloadId);
        }
    }
}

```

图 12-15 在主机端实现并发作用力计算的 pd 方法

每个主机线程都有本地指针 pos 和 chg，分别指向粒子的 position 和 charge。f_i 和 f_j 指向的缓存（用于作用力计算）作为 __computeForce() 的输入。然后，粒子数目 n 和主机线程标识符 offloadId 传入到计算内核中。对于作用力累加，只有变量 f_i、粒子数目 n 以及主机线程标识符 offloadId 是必需的。值得注意的是，对于作用力累加，所有在不同卸载操作中进行分布计算的作用力可通过 f_i 访问（共享内存）。accumulateForce() 的实现在 <http://lotsofcores.com> 可看到。

计算内核：__computeForce()（见图 12-16）：作用力计算的并行算法实现（在一个线程组内）。不同的步骤 (0b)[†]、(1)[†] 和 (2)[†] 在源代码中高亮显示。主机线程标识符 offloadId 和内核内部的线程标识符（通过 omp_get_thread_num() 获得）共同定义了在所有并发卸载操作中的全局线程标识符 id。线程标识符 id 接着用于实现作用力计算 ($F_i = \sum_{j < i} F_{ij}$) 在所有线程中 i 值的平均分配。关于力大小的评估实现包含在函数 f_ij() 中。

类 pd（图 12-12）的其他函数：通过 PD 模拟更新位置和速度，或者进行计算测量之类的内容，在这里并没有列出（见图 12-12 中的“...”）。这是因为讨论它们对于理解的并发卸载操作没有帮助。然而，这些函数的实现代码都可以在 <http://lotsofcores.com> 下载（结合

SIMD 优化的代码，用于基准测试)。

```
// Compute kernel: implements  $O(N^2)$  force computation using Newton's 3rd law.
__attribute__((target(mic))) void __computeForce(const double *pos,
double *chg, double *f_i, double *f_j, int n, int offloadId) {
    memset(f_i, 0, 3*n*sizeof(double)); // step (0b)† of algorithm above
    memset(f_j, 0, 3*n*sizeof(double)); // step (0b)† of algorithm above
    double *pos_x=&pos[0*n], *pos_y=&pos[1*n], *pos_z=&pos[2*n];
    double *f_i_x=&f_i[0*n], *f_i_y=&f_i[1*n], *f_i_z=&f_i[2*n];
    double *f_j_x=&f_j[0*n], *f_j_y=&f_j[1*n], *f_j_z=&f_j[2*n];
    #pragma omp parallel num_threads(THREADS_PER_OFFLOAD)
    {
        int id=offloadId*THREADS_PER_OFFLOAD+omp_get_thread_num();
        double r_ij_x, r_ij_y, r_ij_z, f;
        for(int k=0; k<THREADS_PER_OFFLOAD; k++) {
            int jStart=(k+omp_get_thread_num())%THREADS_PER_OFFLOAD;
            int jIncrement=THREADS_PER_OFFLOAD;
            // Step (1) of algorithm above: distribute i-values evenly among threads
            for(int i=id; i<n; i+=CONCURRENT_OFFLOADS*THREADS_PER_OFFLOAD) {
                int jStop=i;
                for(int j=jStart; j<jStop; j+=jIncrement) { // disjoint j-values
                    r_ij_x=pos_x[i]-pos_x[j];
                    r_ij_y=pos_y[i]-pos_y[j];
                    r_ij_z=pos_z[i]-pos_z[j];
                    f=f_ij(r_ij_x, r_ij_y, r_ij_z, chg, ..); // magnitude of force
                    f_i_x[i]+=f*r_ij_x; f_j_x[j]-=f*r_ij_x; // Newton's
                    f_i_y[i]+=f*r_ij_y; f_j_y[j]-=f*r_ij_y; // 3rd law
                    f_i_z[i]+=f*r_ij_z; f_j_z[j]-=f*r_ij_z; //  $F_{ij} = -F_{ji}$ 
                }
            }
            #pragma omp barrier // step (2)† of algorithm above: else-clause
        }
        // step (2)† of algorithm above: if-clause → accumulate forces  $f_i$  and  $f_j$  into  $f_i$ 
        int chunkSize=(int)ceil((double)(n)/THREADS_PER_OFFLOAD);
        int start=omp_get_thread_num()*chunkSize;
        int stop=start+chunkSize; stop=(stop<n?stop:n);
        for(int i=start; i<stop; i++) {
            f_i_x[i]+=f_j_x[i];
            f_i_y[i]+=f_j_y[i];
            f_i_z[i]+=f_j_z[i];
        }
    }
}
```

图 12-16 根据牛顿第三定律实现作用力计算的内核

12.3.3 性能评估：之前与之后

针对作用力计算算法使用单卸载操作时非常不好的可扩展性，本章开始说明了本章的主旨：并发内核卸载。现在，我们使用多个并发卸载操作，每个卸载操作在协处理器上开启 30 个线程。

图 12-17 说明了同单卸载操作相比，使用不同数量的并发卸载操作所带来的性能提升。

注意 在所有情况下，我们比较了底层 PD 模拟的结果，发现 1000 PD 更新步骤的结果几乎完全一致。这里要注意的是，因为不同的求和顺序以及浮点数操作的不可结合性，结果完全一致是不太可能实现的。

这里的性能指标是每秒发生的粒子间交互的次数（单位为 10^{10} ），简称为“GigaIterations/s”（越大越好）。

对于不同数目的交互粒子，并发卸载方法都要优于非并发卸载方法。随线程组（也就是线程）的增多，其性能几乎实现了线性加速比。对于 64000 个粒子，相对于非并发的情况，实现了 1.25 倍的加速比。这个性能结果也支持了我们利用对称性进行作用力计算和并发卸载操作的最初动机。我们证明了在解决同一个问题时，多个小规模卸载计算的性能可以优于一个大规模卸载计算的性能。

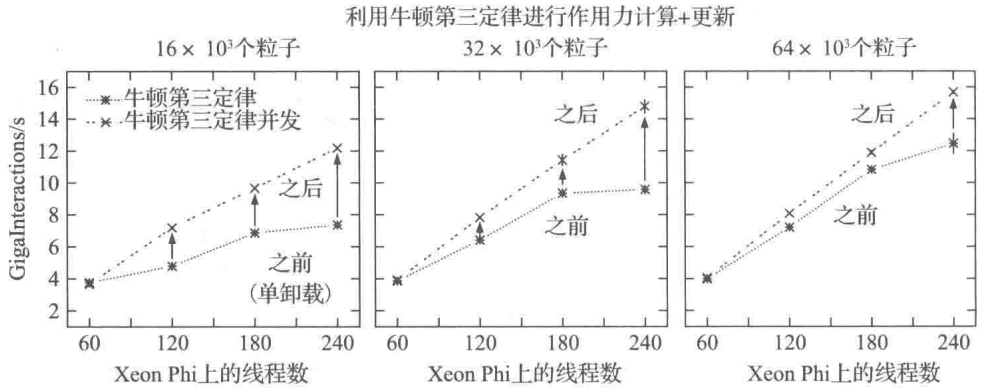


图 12-17 利用牛顿第三定律进行作用力计算时在并发和非并发情况下 PD 模拟的性能。像非并发情况一样，在并发情况下，总共有 8 个主机线程将相同的作用力计算内核卸载到协处理器上。所不同的是，并发情况下每次卸载操作开启了 30 个协处理器线程：对于“120 个线程”的数据点，4 个主机线程并发卸载到协处理器上。1000 个 PD 更新步骤的卸载计算都是在 Intel Xeon Phi 协处理器上进行的

同 CPU 性能的对比：我们利用牛顿第三定律（以及并发卸载操作）实现作用力计算的机制，并不只是针对协处理器（除了卸载的制导指令）。因此，同一份代码在没有协处理的情况下也可以在标准 CPU 上执行。

对于 CPU 基准测试，本章开始时就描述了计算节点的构成。这个计算系统的两个 CPU 插槽跨过两个不同的 NUMA 域，这两个域通过 Intel QPI（Quick Path Interconnect）相连。通过 QPI 的通信（例如，同步操作）是可能降低应用程序性能的。使用两个并发的线程组（每个线程组包含 16 个线程），每个线程组部署到一个 CPU 插槽上，可能提升整个应用程序的性能。

图 12-18 比较了在 Intel Xeon Phi 7120P 协处理和主机（双 CPU 插槽）上执行相同代码时的性能，与在两个 CPU 上统一部署 32 个线程的非并发执行相比，在每个 CPU 上部署大小为 16 的线程组，两个 CPU 并发执行在 16000 个粒子的情况下，实现了 1.5 倍的性能提升；在 64000 个粒子的情况下，实现了 1.2 倍的性能提升。在协处理器上，最高可以实现 2.3 倍的性能提升。

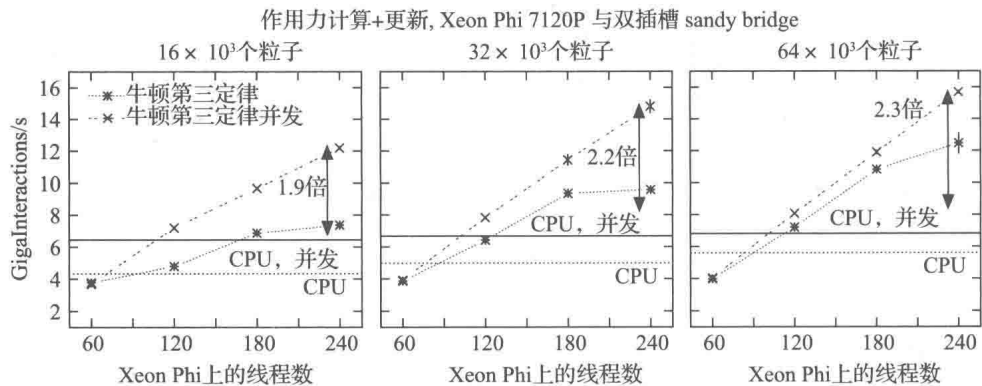


图 12-18 利用牛顿第三定律进行作用力计算时在并发和非并发情况下 PD 模拟的性能。各个分图所要表达的意思和图 12-17 一样。同时，增加了与同双插槽 Intel Xeon 处理器（Sandy Bridge）的性能对比，总共开启了 32 个线程。水平虚线是非并发情况下的性能，水平实线是在每个 CPU 插槽上分别部署两组线程（每组包含 16 个线程）时的性能

通过使用内核并发执行方法提升应用程序性能，这里的 PD 示例证明了这个观点：运行在协处理器上的优化代码，当返回 CPU 端运行时，也可带来性能提升。

12.4 总结

当卸载区域没有暴露出足够的并行性来满足协处理器上的线程需求时，并发内核卸载可能是一个提升总体性能的方法。

当从主机并发卸载多个内核时，下面的方法可提升内核在协处理器上的性能：

线程关联（参见 12.2.1 节）：OpenMP 线程组以及线程与设备的关联在程序运行期间保持不变，除非并行区域的“形状”发生变化。

- 当在主机使用 OpenMP 进行卸载操作时，需要在计算真正开始之前执行一个初始互斥的卸载操作并且在协处理器上创建线程。在这种方式下，线程在卸载区域内的分布取决于参数 MIC_KMP_AFFINITY 和 MIC_OMP_PLACES 值的设定。
- 当在主机程序使用 MPI 进行卸载操作时，需要在协处理上显式指定线程分布。例如，通过使用函数 sched_setaffinity() 或者设定 MIC_KMP_PLACE_THREADS 对协处理器进行恰当的逻辑划分。在设备划分内部，分配给 MIC_KMP_AFFINITY 和 MIC_OMP_PLACES 的参数是有效的。
- **数据传输**（见 12.2.2 节）：当前 COI 的实现为线程提供了主机和协处理间的专门通信通道（卸载模式下的内核调用），但只能使用一条数据传输通道。
- 当在主机应用程序中使用 OpenMP 时，因为只有一条数据通道，并发数据传输会串行化。除此之外，还会对 COI 的内部数据结构产生竞争。当数据包非常小时，对性能的影响就会非常明显。
- 当在主机上使用 MPI 时，每个 MPI 进程都会有自己的数据通道。除非应用程序使用 MPI+OpenMP 混合编程，否则不会存在竞争。
- 如果出现短消息的频繁传输，最好使用少量的 OpenMP 线程，或者转为 MPI，或者转为 MPI+OpenMP 的混合模式。

我们在两个实际用例中使用并发卸载操作：MKL dgemm 计算和 PD 模拟。

对于 dgemm（参见 12.2.1 节和 12.2.2 节），我们得出：只要不超过协处理器能够处理的线程数目，协处理器的整体计算性能会随着并发卸载数目的增加而线性提高。当在主机上使用 OpenMP 时，连续 dgemm 内核卸载间发生的数据传输会降低程序性能。然而，当使用 MPI 时，与没有数据传输相比，几乎没有性能损失。对于执行多次类 dgemm 操作的应用程序，程序员可参考这里的 MKL 示例，研究如何以一种简单的方式为多个小规模或者中等规模的矩阵划分协处理器。

PD 示例展示了如何将 MKL 应用到更加复杂的应用中，以及如何避免并发卸载操作一些不好的特性。代码中使用的内核并发卸载的方式初看起来有些奇怪，但经过一小段时间的思考就会明白：我们将计算根据线程数目 P 进行划分，并划分成 m 个并发卸载操作，每次卸载操作有 P/m 个线程。每次并发卸载操作中，相关线程处理了总计算量的 $1/m$ 。随后，合并所有卸载操作的计算结果。因为随线程数目增多时 PD 计算内核的可扩展性不好，所以只有符合协处理的线程需求，并发方法才能帮助提升程序的整体性能。如果有 8 个并发卸载操作，每个卸载操作开启了 30 个线程，相对于只开启一次卸载操作（共 240 个线程），可以获得 25% 的性能提升。

对于同 PD 示例类似并且可扩展性并不好的应用程序，可以使用本章描述的方法来提升整体性能。例如，一篇关于并发内核执行的论文将并发内核卸载操作整合进热力学模拟代码（见 Wende et al., 2014）。

12.5 更多信息

- Jeffers, J., Reinders, J., 2013. Intel Xeon Phi Coprocessor High Performance Programming, first ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Newburn, C. J., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F., McGuire, R., 2013. Offload compiler runtime for the Intel Xeon Phi coprocessor. In: Supercomputing. Springer, Berlin, Heidelberg.
- Thijssen, J., 2013. Computational Physics, second ed Cambridge University Press.
- Wende, F., Cordes, F., Steinke, T., 2014. Concurrent kernel execution on Xeon Phi within parallel heterogeneous workloads. In: Silva, F., et al. (Eds.), Euro-Par 2014 Parallel Processing-20th International Conference, Porto, Portugal, August 25-29, 2014, Proceedings, LNCS vol. 8632. Springer, pp. 788-799.
- 本章及其他章代码下载地址，<http://lotsofcores.com>。

MPI 和异构计算

Jerome Vienne, Carlos Rosales, Kent Milfeld
USA, TACC

13.1 现代集群中的 MPI

对于许多用户来说，MPI (Message Passing Interface) 是一个在“MPI 任务”间传递数据的“黑盒”。对于程序开发者来说，MPI 的美妙之处在于算法代码同底层硬件无关。然而，在过去几十年里，相对于 MPI 标准诞生的 1993 年，硬件架构已经成为分层架构并且更加复杂。

图 13-1 展示了一个 MPI-1 标准时期的典型计算系统。每个计算节点只包含一个处理器，连接到一个 NIC (Network Interface Card, 网卡)，每个 NIC 连接到交换机上的一个端口。系统中各计算节点的连接都是统一的，应用程序任务在计算系统中所处的位置对通信性能没有影响。

图 13-2 展示了 MPI-3 时代异构高性能计算系统的典型配置。每个节点有多个处理器。每个节点通过分层交换机 (这里是“胖树”拓扑) 连接。在节点内部，每个处理器有多个内核和一个采用众核架构的 Intel Xeon Phi 协处理 (或者 GPU)。同处理器一样，协处理器通过 PCI-e 总线连接到 NIC (实际是上一个主机通道适配器 (Host Channel Adapter, HCA))。

计算节点配备众核协处理器引起了应用程序开发者和用户的关注。通常情况下，一个协处理器包含 60 多个内核，多个处理器包含 10 ~ 40 个内核。如果在每个内核上运行一个 MPI 任务将会在节点内部和节点间 (通过 NIC) 导致通信风暴。同图 13-1 描述的单处理器节点、MPI 单任务执行相比，现代计算节点的通信通道数量多了两个数量级以上。

为了降低通信开销，应用程序通常采用混合技术：利用计算的对称模式，采用 MPI+OpenMP 的混合编程模式 (MPI 进程既可以在处理器执行，也可以在协处理器上执行)。当应用程序的特征适合卸载时，主机上的 MPI 进程会将计算工作移植到协处理器上，以多线程方式执行。这一种方法避免在协处理器上开启进程，从而减少了计算节点上的 MPI 进程数量。然而，这仍然有必要使用 OpenMP、pthread、Intel Cilk Plus 等模型对应用程序进行多线程编程——与 MPI 代码进行高效整合。

在这种思想的指导下，HPC 应用程序需要选择最优的 MPI 进程数目、任务执行地点及实现负载均衡。

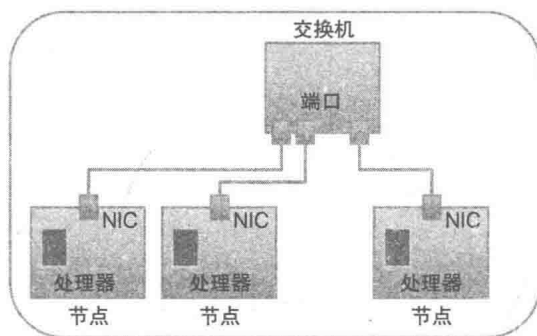


图 13-1 早期 MPI 计算中的单处理器节点

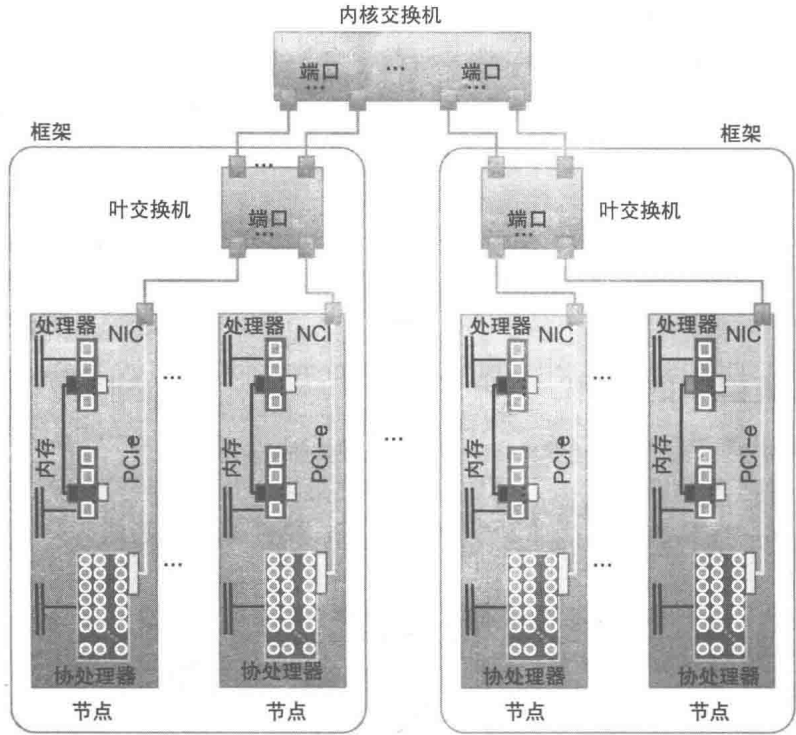


图 13-2 现代 HPC 集群：通过多层交换机相连并包含多个处理器和众核处理器的计算节点

13.2 MPI 任务地点

如图 13-1 所示，在早期的 MPI 编程中，计算节点非常简单，不用考虑 MPI 任务的执行地点。MPI 将读取一个包含主机名称的文件，在每个主机（节点）上初始化一个 MPI 进程（任务）。每个 MPI 进程会根据在进程列表中的位置分配一个唯一的进程号。通过调整进程列表，就会实现进程和硬件处理器之间的不同映射，尽管没有必要这么做，因为对于（单核）处理器而言，这里的通信互联都是对称的。

现代 HPC 系统的每个节点会配备多个处理器、协处理器和图形加速卡（GPU）。每个协处理和图形加速卡都有自己的内存，而处理器间会共享内存。因此，像 Intel Xeon Phi 这样的协处理为 UMA（Uniform Memory Access，非一致性内存访问），而处理器为 NUMA（Nonuniform Memory Access，非一致性内存访问）。同时，并不是所有的处理器和设备都直接和网络接口互连。

在多核系统上，为每个内核启动一个 MPI 任务（进程）是常用方法，这样每个节点会有十几个 MPI 任务（进程）。当所有的内核都被执行通用程序的 MPI 任务（进程）占用时，随机映射是非常好的选择。然而，对于科学计算领域的应用程序而言，相邻任务间存在通信或者通过一个节点上的任务（进程）进程聚集通信。因此，选择一种高效的映射方式会提高通信性能。这会在下面详细解释。

当配置协处理器的系统中，MPI 混合程序很有可能在主机上开启多个 MPI 任务（进程），也会在协处理上或多或少地开启几个 MPI 任务（进程）。OpenMP 并行区域就会为主机和设备的并行区域开启不同数目的线程。所以，在现代 HPC 系统的异构、非对称和 NUMA 架构上运行混合 MPI 程序，任务（进程）和线程与硬件架构的映射就会非常重要。

除非有硬件架构图，否则关于 `cpu-ids` 和内部硬件设备间的映射，不会有太多工作可以做。幸运的是，从大约 2010 年起，一个称为 `hwloc` 的可移植硬件本地化软件包，可检测并且列出 NUMA 内存节点（处理器 + 本地内存）、I/O 设备（如 Infiniband HCA、PCI 桥）等。`hwloc` 不需要 `root` 权限，因此用户可在在自己的用户目录下下载、安装并运行该软件。

图 13-3 展示了 Stampede 系统的架构特性。通过仔细观察图 13-3，可以得到在主机上进行任务（进程）和线程分配所需的详细配置信息。首先，标记为插槽 P#0 和 P#1 的方框为两个 8 核处理器。从 NUMA 的角度看，一个 NUMA “节点” 就是处理器及其对应的内存。注意，给 NUMA 节点指定的“16GB”内存为处理器的本地内存。芯片上的处理器及其对应的分层存储器被标记 NUMANode P#0 与 P#1（插槽是 HPC 领域的常用名称，指处理器及其主板上的支撑结构）。在下面的讨论中，我们将交替使用 NUMA 节点和插槽。

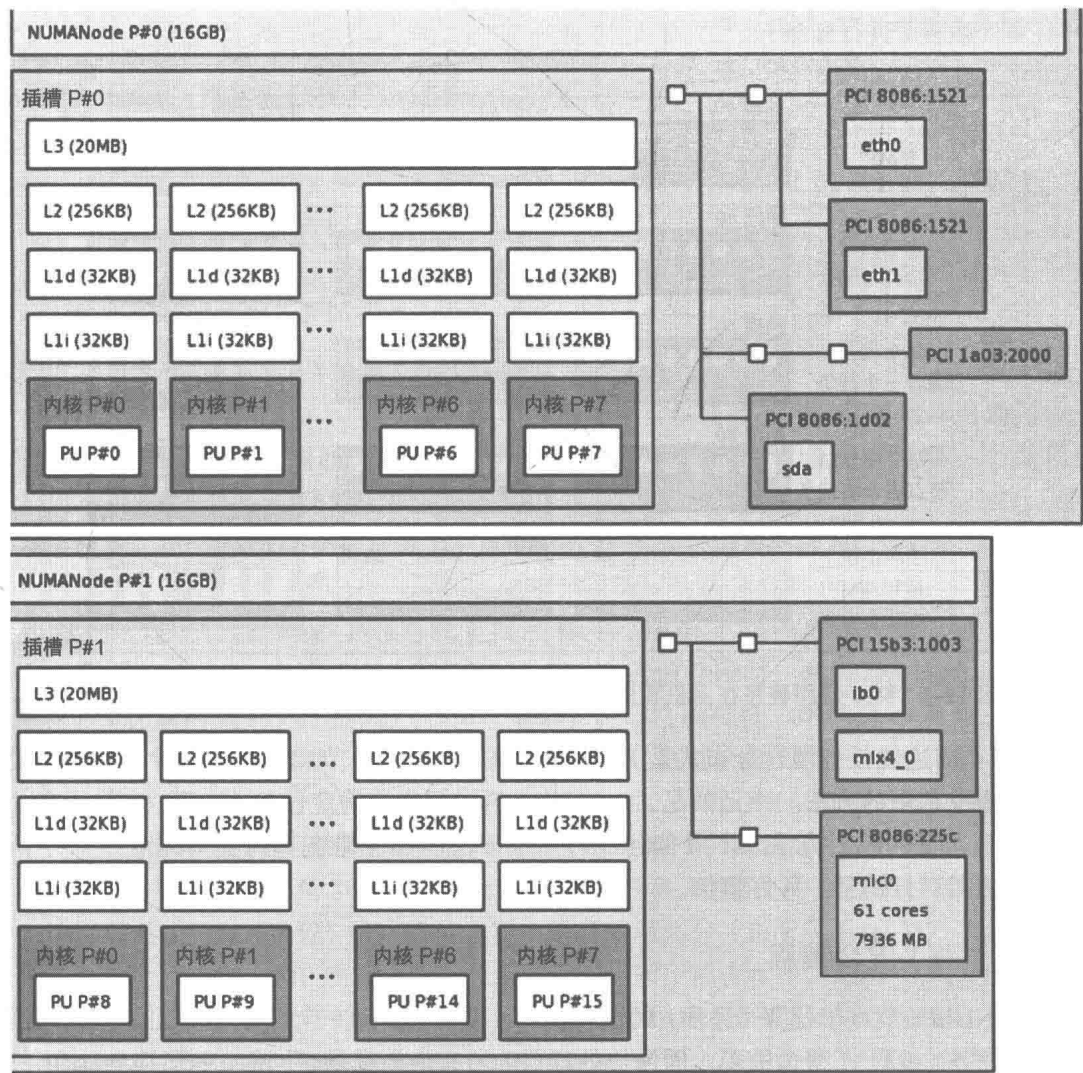


图 13-3 Stampede 节点的硬件架构图（hwloc）

插槽内部的内核被标记为 PU（Processing Unit，处理单元），如插槽 0 上的 P#0 ... P#7 和插槽 1 上的 P#8 ... P#15。如果开启超线程技术，每个深灰色的方框中将会有两个 PU。这

些 PU 标记会被 BIOS/OS 设置为硬件索引。处理器信息也记录在 `/proc/cpuinfo` 文件中，可用于任务（进程号 rank）和线程（线程 id）与处理器的映射。

每个 NUMA 节点的右边是设备互联。连线为 PCIe 通道，小白框为桥。在 #0 NUMA 节点，#0 插槽中处理器的 PCI 总线连接网络（eth0/1）和硬盘（sda）。在 #1 NUMA 节点，#1 插槽连接 InfiniBand HCA（mlx4_0），Intel Xeon Phi 协处理器（mic0）连接 PCI 8086:225c 控制器。

一旦了解了设备的架构布局，你就会对 MPI 任务（进程）和线程的分配以及与硬件的映射做出明智的选择。

图 13-4 对 Stampede 系统中任务在每个计算节点的处理器和协处理器上的分布情况进行了分类和说明（白色圆圈为混合线程）。如之前提到的，除非只有少量的通信并且每个任务在协处理器上的通信存储非常小，否则纯 MPI 的方法是不可行的。同样，只在主机使用 MPI（+ 卸载），不需要协处理执行任务的情况，本节不讨论（尽管混合方法在主机上使用 MPI 任务（线程）进行卸载）。

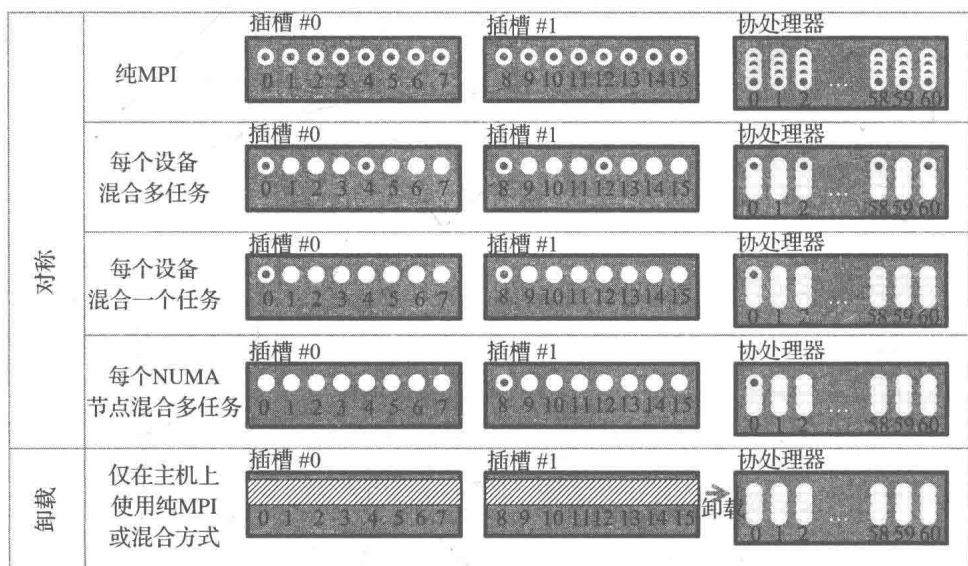


图 13-4 MPI 应用程序在主机和协处理器上的常用任务分布方法（Stampede 节点）

图 13-4 描述的任务混合分布式是 HPC 应用程序在进行单节点和多节点计算中可以使用的合理及常用的方法。幸运的是，Intel MPI 可以智能地确定任务（进程）和线程在处理器和协处理器上的映射方法。不管如何，有一个方法能够测量在 MPI 的非最优实现中低效任务分布的相对开销是一件好事情。

单任务（进程）混合编程

每个 NUMA 节点单任务（进程）的混合模式（主机上启动一个任务（进程），协处理器上启动一个任务（进程））最为简单。因为协处理器的所有内核都是相同的（除了第 60 个内核会进行守护进程的操作之外），所以这个单任务（进程）可映射到协处理器的任意内核上。内核间进行线程映射的三种模式为：compact、balanced 和 scatter。不同映射方式将通过一个示例进行解释。假如我们有一个计算系统，该计算系统上配置了 4 个内核，每个内核上开启了 4 个硬件线程。如图 13-5 所示，我们在系统上分别按照 compact、balanced 和 scatter 方式开启了

8 个线程。在 scatter 模式下，从第一个内核开始，按照循环方式在每个内核上分配两个线程。balanced 模式下采取和 scatter 模式下相同的策略，但是每个内核上分配的线程索引是连续的。在这种情况下，compact 模式会导致有些内核处于空闲状态。正是由于这个原因，当开启的线程数量少于内核数时（在这种情况下是内核 *4。——译者注），推荐使用 scatter 和 balanced 模式。

对于单任务（进程）的情况，当在并行区域开启线程时，环境变量 KMP_AFFINITY 用于设置 OpenMP 运行时在协处理器和处理器上进行线程分配（关联）的方式。语法是：

```
export KMP_AFFINITY=[<modifier>] type
```

其中，三种线程分配（关联）的方式是 compact、balance 和 scatter，并且可以使用 modifier 中定义的处理器列表来显式将线程关联到特定内核上。除了使用处理器列表显式绑定线程外，在协处理器上，可以通过使用粒度和内核修改器将线程固定到一个内核上的一个硬件线程上，或者允许这个线程在所有的硬件线程中“悬空”。

对于协处理器，使用这三种分配方式的其中一种是有意义的。当对这些关联方式进行试验时，一个比较好的开端是尝试：

```
KMP_AFFINITY="granularity=core,balanced"
```

在主机端，相同的 KMP_AFFINITY 设置可用来进行单任务执行。任务顺序可通过一个偏移量来设置（线程 0）。例如，设置 KMP_AFFINITY = compact,0,8 将会把第一个线程或者是 MPI 任务（进程）分配到第 8（默认为 0）个内核上执行。

在主机端，Intel MPI 将会把计算节点的第一个任务分配给直接和 HCA 相连的处理器内核上。对于在主机运行单任务的应用程序，这将保证最好的通信性能。

如图 13-5 说明，PU #1 的 PCI 总线直接和 HCA 相连（不是一个常用的设置）。因此，第一个进程被 Intel MPI 映射到 PU#1。当进行点对点通信性能时，映射到 PU #0 的 MPI 任务（进程）要比映射到 PU #1 的 MPI 任务（进程）通信性能要低。当消息小于 4KB 时，性能会降低 40%；当消息长度介于 8KB 与 16MB 之间时，性能会低 20% ~ 1%。

当在主机或者协处理器上开启多个任务（进程）时，分配任务（进程）以及提供合适用于每个任务映射的核或者硬件线程更具有挑战性。Intel MPI 的默认分配是合理的。例如，如果两个任务（进程）被分配到 Stampede 主机上，每个任务（进程）将会被映射到不同的处理器上，这样只会在任务（进程）所在的处理器上进行线程划分（fork）。Intel MPI 有许多调整 MPI 任务（进程）和 OpenMP 线程位置和分布的机制。对于混合程序，KMP_AFFINITY 用于定义 MPI 任务（进程）的线程的分布，I_MPI_PIN_DOMAIN 用来定义每个 MPI 任务（进程）用来开启其线程所在的域（每个域包含一系列内核）。例如，当在每个 NUMA 内存节点（共有两个 Stampede 节点）各分配一个 MPI 任务（进程）时，设置 I_MPI_PIN_DOMAIN = socket 将确保 MPI 任务（进程）分配到不同的处理器上。

如果应用程序需要对每个域的内核进行精细选择，可将 I_MPI_PIN_DOMAIN 设定为一个标记列表（mask list），列表中的每个标记元素代表域中的一个内核。把 MPI 任务（进程）

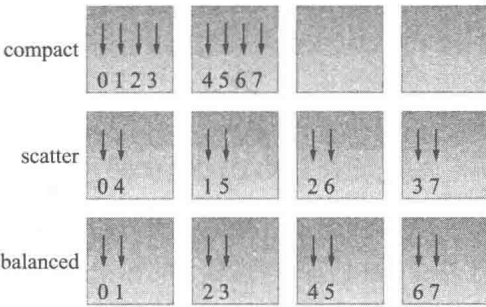


图 13-5 KMP 关联方式。在主机上，scatter 和 compact 在插槽间以类似的方式进行线程关联

顺序分配给各个元素。也就是说，对于一个计算节点，把索引号最小的 MPI 任务（进程）分配给第一个域，下一个分配给第二个域。例如，如果 `OMP_NUM_THREADS = 3`，计算节点上分配了两个 MPI 任务（进程），用户希望索引为 0 的任务将它的三个线程运行在索引为 8、9 和 10 的内核上，索引为 1 的任务将它的三个线程运行在索引为 0、1 和 2 的内核上。标记列表应该如下：

```
标记元素 1: 0000 0111 0000 0000
标记元素 2: 0000 0000 0000 0111
```

为方便起见，标记列表可用十六进制数赋值，每个标记列表用方括号括起来。那么，对于两个 MPI 任务（进程）的标记列表可如下表示：

```
export I_MPI_PIN_DOMAIN="[ 1500,15 ]"
```

专家会避免针对特定厂商的硬件特性设置线程关联，而是通过设置环境变量将信息传递给运行时。在 UNIX 系统上，会在代码中使用 `sched_setaffinity` 和 `sched_getaffinity` 来将任务线程（每个线程有一个唯一的 id）绑定到给定的内核上。

13.3 DAPL 提供者的选择

选择正确的提供者列表有助于提升通信性能。本节将会介绍各提供者的性能，然后说明如何通过 `I_MPI` 环境变量来指定提供者。

Intel MPI 能够自动选择提供者列表，然而，这个选择并不总能保证提供最佳性能。提供者的选择可以通过环境变量 `I_MPI_DAPL_PROVIDER_LIST` 直接控制。这个变量的值是逗号分隔的提供者列表（最多三个），语法如下：

```
I_MPI_DAPL_PROVIDER_LIST=\  
<Provider1,Provider2,Provider3>
```

列表中的每个提供者都扮演着不同角色：

- **Provider1**：针对短消息。该提供者应该有低延迟性能。
- **Provider2**：针对节点内的长消息。适用于涉及协处理器的 MPI 程序或者使用 DAPL 进行节点内通信。
- **Provider3**：针对节点间的长消息。

在 Stampede 系统上，提供者列表的值为：

```
I_MPI_DAPL_PROVIDER_LIST=\  
ofa-v2-mlx4_0-1u, ofa-v2-scif0, ofa-v2-mcm-1
```

13.3.1 第一个提供者 OFA-V2-MLX4_0-1U

提供者 1 由 Intel MPI 自动设定，不需要任何用户的操作。Stampede 系统直接使用节点上可用的 InfiniBand 卡进行消息交换。

13.3.2 第二个提供者 ofa-v2-scif0 以及对节点内部结构的影响

第二个提供者 `ofa-v2-scif0`（称为 `scif`）对节点内长消息性能有着重要的影响。同时，节点内部结构的选择对 `scif` 的效果也有着重要的影响。这个结构可通过使用如下环境变量选择：


```
I_MPI_FABRICS=\n<fabric>|<intra-node fabric>:<inter-nodes fabric>
```

基于 InfiniBand 集群进行节点间通信的推荐结构是 DAPL。Intel MPI 默认使用共享内存进行节点内通信。然而，如果选择 DAPL 进行节点间和节点内通信，很多 MPI 应用程序的性能会得到提升。这就意味着在设置环境变量 I_MPI_FABRICS 时，有两个不同的选择：

```
(I_MPI_FABRICS = dapl)\n或者\n(I_MPI_FABRICS = shm:dapl)
```

后者会根据 MPI 应用程序执行时的消息大小进行选择。

为了说明不同选择对性能的影响，图 13-6a 提供了 DAPL（使用和没使用 scif）与 shm : dapl 的性能对比。使用 Intel MPI 5.0 在两个 CPU 间进行节点间通信。从图 13-6b（协处理器间）和图 13-6c（CPU 和协处理器间）可以观察到相似的结果。这些结果通过运行 Intel MPI Benchmark (IMB) 4.0 中的 Pingpong 测试程序获得。

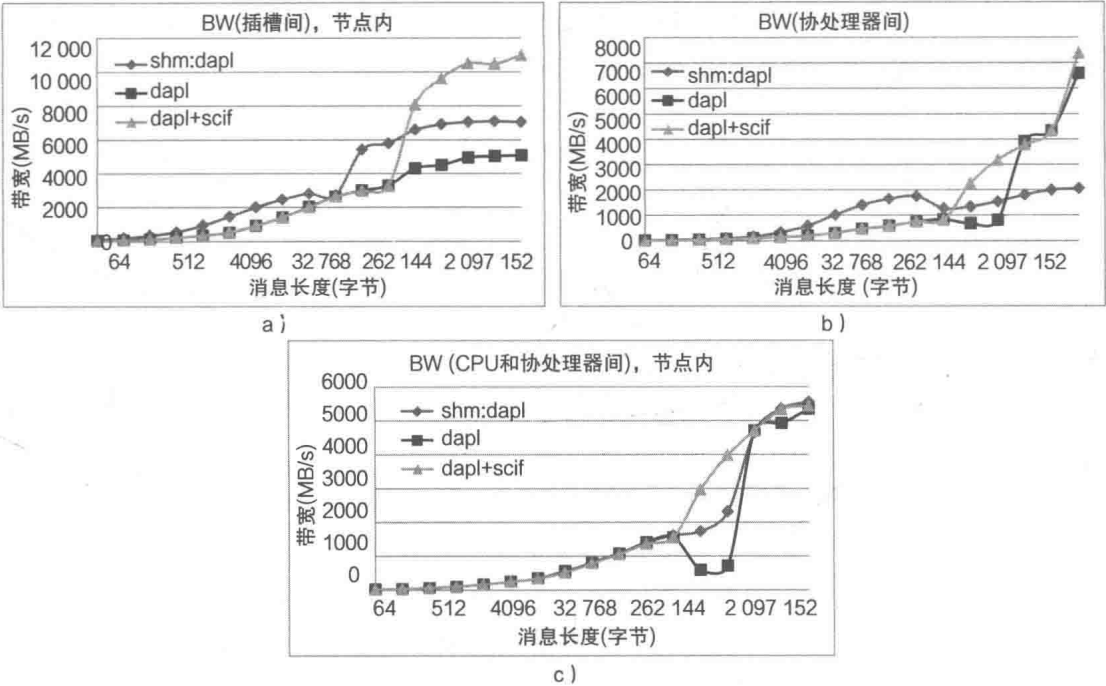


图 13-6 MPI 提供者的性能：a) CPU 间通信；b) 协处理器间通信；c) CPU 和协处理器间通信

可以得出，shm:dapl 可在消息较短时提供最高性能。当应用程序传递的消息超过 256KB 时，dapl+scif 是最佳选择。我们也可以发现 scif 没有被 Intel MPI 自动应用于 CPU 间或者协处理间的通信。这些情况下的最高性能都是通过显式指定最佳组合获得的。这可能是 Intel MPI 在未来需要改进的地方。

在所有情况下，当选择 shm:dapl 时，第二个提供者对性能没有影响。

13.3.3 最后一个提供者[⊖]

当使用 InfiniBand 网络接口时，Intel Xeon E5 处理器提供的有限 P2P 本地读取带宽会

[⊖] 也称为代理。

明显影响与 Intel Xeon Phi 协处理器的通信性能。如图 13-7 所示，P2P 本地读取操作的低带宽是 MPI 库的性能瓶颈。

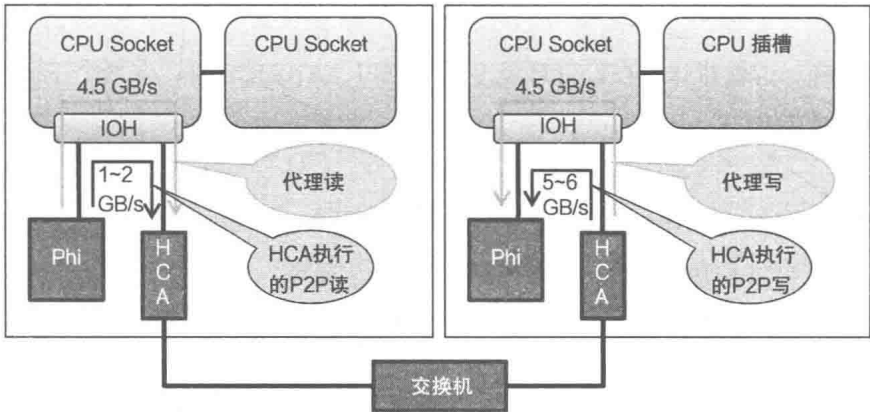


图 13-7 本地 MPI 读 / 写性能

为了解决这个问题，Intel MPI 围绕这个限制引入了一个增强设计，为 Stampede 及其相似系统的主机和协处理器间的 MPI 消息传递提供了更好的性能。

现在，Intel Xeon E5 处理器家族可以在 CPU 和 PCI 设备间实现最高 7GB/s 的读取带宽，写入带宽也差不多。同样，使用单口 InfiniBand FDR HCA，节点间主机和主机的通信带宽也可达到 6GB/s。

对于协处理器和其他节点之间的长消息通信，提供者使用的机制是一个基于主机的方法，称为代理。该代理利用两条通道（协处理器到主机、主机到 HCA），性能为有限的 P2P 本地读取带宽。图 13-7 显示了本地 P2P 读取的低带宽。

该代理对于包含协处理器的节点间通信默认是启动的。但为了说明这个代理有多么重要，我们分别在开启和关闭该代理的情况下使用 Intel MPI 5.0 测试了节点间协处理器到协处理器的通信性能。IMB 4.0 的 Pingpong 测试用例用来测试性能。当较低的读取性能问题通过代理解决后（图 13-8 所示），4.5GB/s 的代理带宽低于我们能够获得的性能（如果本地读性能同本地写性能同样高）。

13.3.4 混合程序的可扩展性

当使用科学计算应用程序时，如何将微型基准测试的运行结果转化为可量化的效果是非常重要的。本章以格子玻耳兹曼方法 (lattice Boltzmann method, LBM) 作为对象研究这个问题。该算法的具体实现对应 lbs3d-mpi, 在 mplabs 包中的开源代码。13.8 节有更多细节

LBM 是一个在高层次细节上研究流体动力学的介观技术。LBM 与传统计算流体动力学的区别是没有使用线性解法器。规则的笛卡儿网格和完全明确的时间演化使得代码并行和

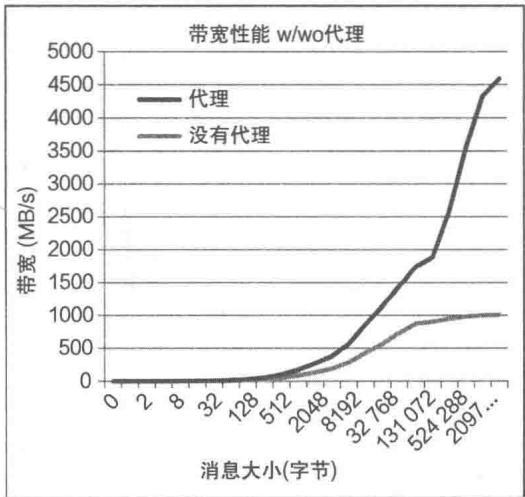


图 13-8 Intel MPI 开启和关闭 InfiniBand 代理时的性能

优化相对简单。通过在每个计算网格点进行一系列的本地计算完成该问题的动态演化，随后通过内存操作将计算结果传给邻居网格点。这就是众所周知的 LBM 的碰撞流（collision-stream）实现方法。该技术更加详细的介绍可阅读 SauroSucci 写的书（见 13.8 节）。

lbs3d-mpi 代码基于自由能（Free-Energy）实现（作者为 Zheng、Shu 和 Chew）。该代码使用了一个基于 D3Q19 分解的底层模型来计算流体阶段的速度和 D3Q7 分解，这将在每个时间步长的每个 MPI 任务（进程）的子域内导致相邻 MPI 任务（进程）通过 6 个面和 12 条边发生通信。图 13-9 显示了 D3Q7 和 D3Q19 的离散速度。为了更加清晰，图 13-9 仅仅对 D3Q7 相关的箭头进行了编号。

为了适应 MPI 交换在每个任务计算域中使用单个 ghost 层。lbs3d-mpi 目前的实现没有将计算和通信重叠。因此，这应该是在最坏情况下研究 Intel MPI 代理对应用程序整体性能影响的一个好例子。

本节使用的代码实现基于 MPI 域分解，并将主要循环使用 OpenMP 进行并行化。要在协处理器上获取高性能，混合实现是必需的。为了最大化每个线程的工作量，在计算的最外层循环中进行了 OpenMP 并行化。把计算域平均分配到所有 MPI 任务（进程）中，计算域划分可由用户控制。本节呈现的例子沿 y 方向进行划分，以便 x 方向和 z 方向上的循环可分别利用向量化和 OMP 并行等优化方法。在 OMP 并行区域没有 MPI 调用。

由于算法的特性，LBM 模拟倾向使用相当大的计算网格，MPI 通信时间通常由在各个子域面进行交换的长消息数据决定。当使用混合实现时，由于每个任务需要传递更大的消息数据，这种效应更加明显。因为代理对大数据交换是最有效的。所以当这份代码使用协处理器的本机模式 and 对称模式时，开启 Intel 基于代理的通信，应该在程序的可扩展性方面有很大的不同。

图 13-10 展示了开启代理对来自 mplabs 的 lbs3d-mpi 代码整体性能的影响。性能指标采用每秒更新的格子数（单位为 10^6 ，MLUPS）：

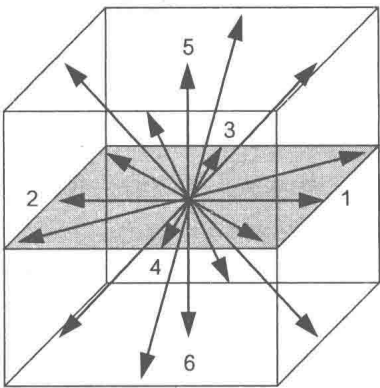


图 13-9 D3Q19 模型的速度分解方向。
注意，数据必须在计算域的所有面和边上进行交换

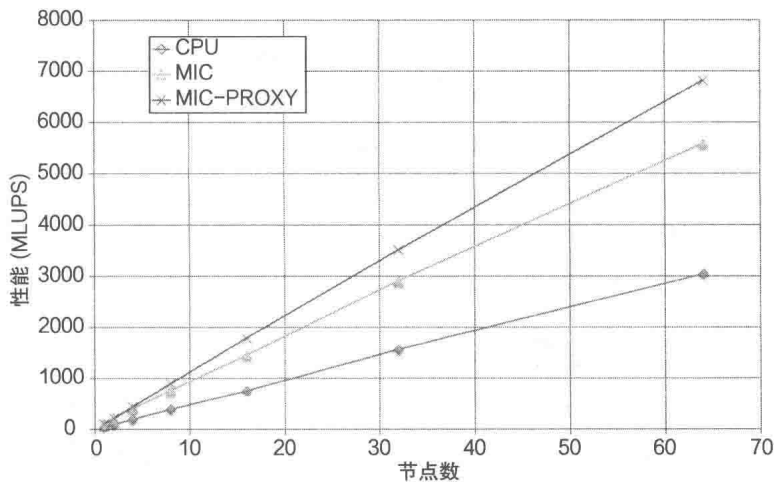


图 13-10 LBM 代码在 CPU 和协处理器上的可扩展性：开启代理、未开启代理

$$\text{MLUPS} = (\text{网格容量} \times \text{时间步长}) / (\text{执行时间} \times 10^6)$$

初始实现使用了 240 个 OMP 线程，并设置线程关联为 compact 模式，每个节点分配一个 MPI 任务（进程）。计算域在每个节点保持 $240 \times 240 \times 240$ 个网格点。在这个例子中，MPI 通信由超过 1MB 的消息交换决定。测试中开启代理并将环境变量设置为：

`I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-mcm-1`

图 13-10 非常清晰地显示了使用代理对任何涉及协处理器的通信带来的优势。带方框的线展示了 CPU 初始实现版本的可扩展性，当使用代理后并没有发生变化。带十字和三角形的线分别代表了使用和不使用代理的性能。当运行 64 个节点时，最多获得了 22% 的性能提升。请注意，这和通信时间的大幅缩减是成比例的，因为 MPI 仅仅是整体运行时间的一部分。

代理对 lbs3d-mpi 代码整体性能的效果，在使用对称模式时也可以得到相似的结果，并且对性能的影响更加明显：当在 64 个节点上执行时，获得了 31% 的性能提升。因为代码在协处理器上的运行性能大约是在主机处理器上运行性能的两倍，所以每个节点使用了三个 MPI 任务（进程），一个分配给主机处理器，两个分配给协处理器。这虽然实现了一个静态负载均衡，然而，每个计算节点的 MPI 任务（进程）增长了 3 倍，并增加了通信。

13.3.5 负载均衡

正如前文提到的，因为这份代码使用了域分解方法，并且在协处理器上的性能是在双插槽 CPU 上性能的两倍（CPU 主机上开启了 16 个线程，协处理器上开启了 240 个线程），协处理器上部署的 MPI 任务（进程）也是处理器的两倍。这是一个实现静态负载均衡的简单方法。注意，如图 13-11 所示，若不如此处理会导致性能降低。

13.3.6 任务和线程映射

值得注意的是，对于所有的代码版本，MPI 任务映射都使用了如下设置：

```
I_MPI_PIN_MODE=omp:compact
KMP_AFFINITY=compact,granularity=fine
```

虽然我们可以设置 MPI 任务（进程）和 OMP 线程映射与关联，但是必须要小心，要避免将多个逻辑线程绑定到主机的相同内核上或者协处理器的相同硬件线程上。在最坏的情况下，所有线程可能会在同一个内核或者同一个硬件线程上执行，从而导致多个线程的串行执行。作为一个例子，将所有线程都绑定到主机索引为 0 的内核上，仅仅获得 4MLUPS 的性能，比正确绑定的性能（大约 50MLUPS）低了一个数量级。在协处理器上这个效果更加明显，性能仅为 1.4MLUPS，比最佳映射慢了两个数量级（大约 112MLUPS）。

通过 Intel MPI 库选择的绑定会使用硬件信息、合理的线程分配机制，通常会提供最高的整体性能。当同其他 MPI 实现一起使用并分析混合代码的性能时，牢记硬件配置和分布模式是非常重要的。

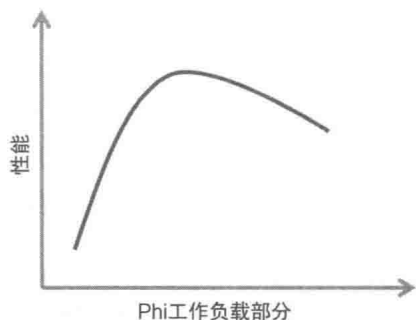


图 13-11 协处理器工作负载部分的性能模型。注意，将太多或者太少工作分配给协处理器会导致性能的降低

13.4 总结

近年来, MPI 实现进行了很多改进以适应更加复杂的计算节点架构: 多层 NUMA 架构、非对称性链接和异构微处理器(处理器和协处理器)。当任务数比内核数少时, 任务和线程在内核上的不同映射会在性能方面产生很大的不同。

为了能够在不同消息大小的情况下都实现高性能, 有必要指定合适的 MPI 提供者。正如本章的例子所示: 使用 Intel MPI 提供的代理在多个协处理器间获得最佳通信性能是非常关键的。

13.5 致谢

Stampede 受到了美国国家科学基金会(NSF)的资助: ACI-1134872。XSEDE (Extreme Science and Engineering Discovery Environment) 也得到了 NSF 的资助: ACI-1053575。

13.6 更多信息

- Goglin, Brice Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality(hwloc). In Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014), Bologna, Italy, July 2014. hwloc, <http://www.open-mpi.org/projects/hwloc/>.
- <https://software.intel.com/en-us/articles/intel-mpi-benchmarks>.
- mplabs code, <https://github.com/carlosrosales/mplabs>.
- Ramachandran, Vienne, J., Van Der Wijngaart, R., Koesterke, L., Sharapov, I., 2013. Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi. In 6th International Workshop on Parallel Programming Models and Systems Software for High-End Computing(P2S2), in conjunction with ICPP, Lyon, October 2013.
- Rosales, Porting to the Intel Xeon Phi: Opportunities and Challenges, XSCALE13, Boulder, CO,USA.
- https://www.xsede.org/documents/271087/586927/CRosales_TACC_porting_mic.pdf.
- Succi, S, 2001. The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Oxford University Press, Oxford, UK.
- Zheng, H.W., Shu, C., Chew, Y.T., 2006. A lattice Boltzmann model for multiphase flows with large density ratio. J. Comput. Phys. 218, 353-371.
- TACC Stampede 系统是具有 6400 个节点的计算机集群, 每个节点包含两个 8 核的 Intel E5-2680 至强处理器和一个 61 核的 Intel Xeon Phi SE10P 协处理器。节点间通过 FDR InfiniBand 网络连接。更多详情请访问: <https://www.tacc.utexas.EDU>。
- 本章介绍的所有测试均使用如下工具链: MPSS V2.1.6720、Intel 编译器 V14.0.1.106 和 Intel MPI V4.1.3.049。

Intel Xeon Phi 协处理器功耗分析

Claude J. Wrigh

美国, Intel 公司

14.1 功耗分析

本章将介绍对 Intel Xeon Phi 协处理器进行功耗分析的方法和工具, 工作负载将选用高性能计算 (High Performance Computing, HPC) 程序。

功耗定义为单位时间消耗的能量, 单位是焦耳每秒 (J/s) 或更常用的瓦特。本章将使用瓦特作为功耗测量的标准单位。

瞬时功耗的计算公式基于欧姆定律: 功耗等于电流乘以电压。电流是通过导体的电子流, 单位是安培。电压等于在静电场中移动单位电荷所做的功, 单位是伏特。

功耗可以由安培数乘以电压来计算, 也可由很多方法来测量, 例如, 可以测量瞬时原始功耗、平均功耗、峰值功耗、最小功耗、均方根功耗或者对原始功耗数据进行移动平均, 本章选用称作“1 秒移动平均” (1-second moving average) 的标准。选用这种功耗分析标准是由于它是热相关的, 与现实世界可测量的热事件具有相互关系。例如, 如果仅测量瞬时功耗, 可能会测得一些持续时间很短的功耗尖峰, 但这些尖峰并不会对硅片温度造成可测量的影响。在实际应用中, Intel MPSS 控制面板显示的功耗就是 1 秒移动平均。

在电子学中另一个常用的功耗术语是热设计功耗 (thermal design power, TDP), 它也使用瓦特来描述。TDP 指的是设备在进行典型操作中产生的最大热量, 制冷设备必须将这些热量驱散。例如, 一些型号的 Phi 协处理器 TDP 是 300W, 还有一些是 245W。TDP 也可作为全速运行该协处理器时功耗预算的基准值。但该协处理器功耗并不会严格低于额定 TDP, TDP 不是潜在的峰值功耗, 而是运行典型应用的额定功耗。精心设计的程序 (比如功耗病毒) 也可能超过额定 TDP, 但典型的 HPC 工作负载绝不会出现这种情况。

测量直流 (direct current, DC) 功耗的方法之一是使用测量电阻。这种电阻阻值很小, 当电流通过电阻时, 将产生与电流值成正比的较小电压, 由此可测得电流的安培值, 再乘以电压值即得到瞬时原始功耗。电压是通过将测量仪器探头置于电源正负极来测量的。这些工作通常由一些数据采集硬件来完成, 比如示波器、数据采集 (Data Acquisition, DAQ) 单元, 甚至简单的微控制器。实际使用中, Phi 协处理器包括一个内嵌的简单功耗分析器, Intel MPSS 工具可以获取该分析器的数值, 如图 14-1 所示。

Phi 协处理器利用电路板上一个叫作系统管理控制器 (system management controller, SMC) 的微处理器。SMC 的工作之一是监控协处理器的输入直流功耗、协处理器的硅片温度传感器及图 14-2 所示的其他热传感器。图 14-2 也展示了补充的 2×3 及 2×4 12V 电源连接器位置 (这两个电源连接器是正常操作所必需的)。如果协处理器使用有源风扇, SMC 也监控风扇速度。基于软件的功耗分析器将利用 SMC 的这些特征获取协处理器功耗和硅片温度。图 14-2 还展示了其他与协处理器 CPU 温度无关的一些热传感器, 比如存储器、入口温

度和出口温度传感器。这些额外的热传感器在服务器或计算节点运行一些高功耗的 HPC 工作负载时比较有用，因为运行高功耗的负载需要对服务器或计算节点进行热调整，比如调整最低服务器风扇速度，甚至调整数据中心的周围温度以达到理想的冷却条件。

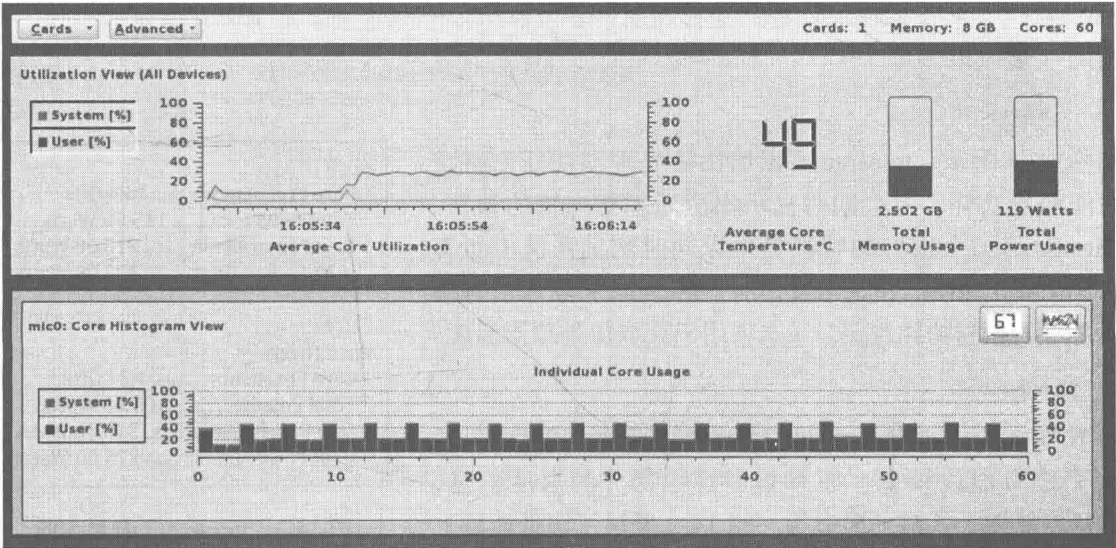


图 14-1 Intel MPSS micsmc 工具显示的功耗和温度

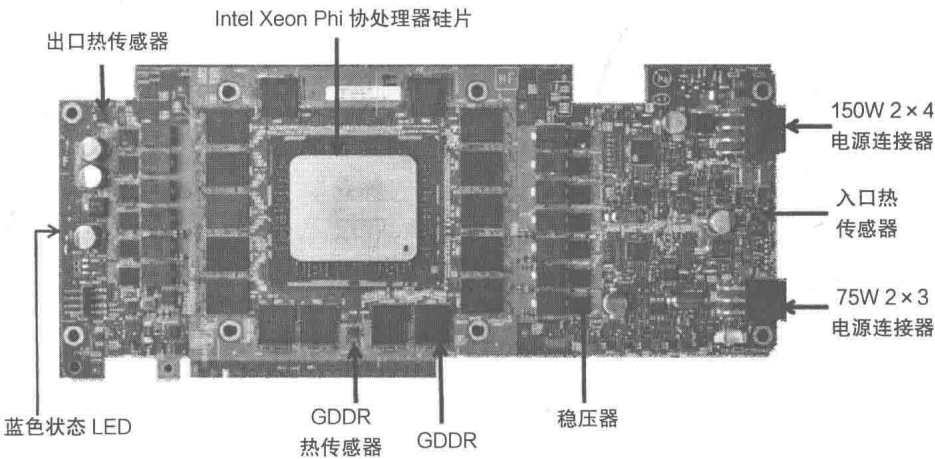


图 14-2 Phi 协处理器卡的热传感器位置

14.2 用软件测量功耗和温度

如图 14-1 所示，标准 Intel MPSS 控制面板的 micsmc 功能支持对协处理器总功耗和硅片温度的监控。如果 Intel MPSS 是按默认用户指南安装的，micsmc 应该已在用户路径下，本章将讨论的使用方法基于 Linux 上默认安装的 Intel MPSS，通过执行下面的命令即可使用 Intel MPSS 控制面板：

```
$ micsmc
```

执行后，Intel MPSS 控制面板就会弹出，不断显示出监控到的协处理器功耗、温度以及协处理器和存储器的使用情况。micsmc 接受参数输入，如果输入下面的命令：

```
$ micsmc -a
```


命令行终端将会显示所有协处理器功耗、热及使用情况指示器。输入下面的命令会显示所有 micsmc 命令：

```
$ micsmc --help
```

micsmc 工具对于监控协处理器功耗和温度非常有用。但如果只想查看协处理器的功耗，需使用 -f 选项：

```
$ micsmc -f
```

如图 14-3 所示，micsmc 可以给出协处理器在任何特定时间的总功耗，也会同时显示出协处理器内核的频率。micsmc 另一个很好的特征是可以同时测量系统中所有的 Phi 协处理器。在请求发出时，micsmc 将获取所有协处理器的瞬时功耗快照。-t 选项用于获取协处理器温度

```
$ micsmc -t
```

如图 14-4 所示，micsmc 可以显示协处理器的温度信息，除了 CPU 温度信息，还有很多其他部分的温度信息，包括存储器、入口 / 出口等。图 14-2 展示了协处理器上这些温度传感器的位置。micsmc 的这一特征在进行平台调整及理想最低风扇速度设定时非常有用，为主机服务器或计算节点设定最低风扇速度可以使风扇功耗最小化。micsmc 工具可以显示系统中或一个集群节点中所有 Phi 协处理器的功耗和热信息。

14.2.1 创建功耗和温度监控脚本

上一节介绍了 Intel MPSS micsmc 功能可以显示协处理器的功耗和温度信息。当用户需要监控关键 HPC 应用的功耗和热信息以及执行平台热调整时，这非常实用。因为 micsmc 运行在主机服务器或计算节点上，所以用户可以使用标准脚本创建基于 micsmc 的简单而实用的功耗及热监控器。

一个实用的脚本应该仅监控协处理器功耗及 CPU 温度，并包含一个时间戳，而不是读出所有的温度值。然而，如之前的例子所示，micsmc 输出信息远比协处理器的总功耗和 CPU 温度要多。如果想仅输出如图 14-5 所示的功耗和热信息，可以使用图 14-6 所示的功耗及热监控脚本。

图 14-6 中的代码在 shell 脚本文件 power_monitor.sh 中（本书代码在 lotsofcores.com 下载）。运行脚本时，屏幕将显示类似图 14-5 的信息，这个方便的小脚本可以运行在后台，来监控协处理器功耗和硅片温度。输出信息可以直接写入文本文件中。

mic0 (freq):	
Core Frequency: 1.24 GHz
Total Power: 115.00 Watts
Low Power Limit: 315.00 Watts
High Power Limit: 375.00 Watts
mic1 (freq):	
Core Frequency: 1.24 GHz
Total Power: 109.00 Watts
Low Power Limit: 315.00 Watts
High Power Limit: 375.00 Watts

图 14-3 micsmc 工具显示的协处理器总功耗

mic0 (temp):	
Cpu Temp: 66.00 C
Memory Temp: 46.00 C
Fan-In Temp: 39.00 C
Fan-Out Temp: 46.00 C
Core Rail Temp: 46.00 C
Uncore Rail Temp: 47.00 C
Memory Rail Temp: 47.00 C
mic1 (temp):	
Cpu Temp: 64.00 C
Memory Temp: 44.00 C
Fan-In Temp: 35.00 C
Fan-Out Temp: 44.00 C
Core Rail Temp: 43.00 C
Uncore Rail Temp: 44.00 C
Memory Rail Temp: 44.00 C

图 14-4 micsmc 工具显示的协处理器温度信息

Timestamp: 11:00:18.576	
mic0 Power:	101.00 Watts
mic1 Power:	107.00 Watts
mic0 Temp:	37.00 C
mic1 Temp:	39.00 C

图 14-5 抽样功耗和温度记录器输出


```
#!/bin/bash

while ; do

    # Build the Timestamp string
    STR=`echo $(date +%N) | sed 's/^0*//'^`
    STR=`printf "%03d" ${((STR/1000000))}`
    STR=`echo $(date +%H:%M:%S)".$STR`
    TIME="Timestamp: $STR"

    # Format the power and thermal data string as desired
    DATA="$({ micsmc -f; micsmc -t; sleep .1; } | grep -e Cpu -e temp -e
Total -e freq | paste -d ' ' - | awk '{print $1,$4,\"\\t\",$6,$7}')"

    # Display our power and thermal data strings
    clear

    echo "$TIME"
    echo "$DATA"

done
```

图 14-6 协处理器功耗和温度监控器 bash shell 脚本 [power_monitor.sh]

注意 使用 Intel MPSS micsmc 工具测量功耗时值得注意的是，Intel Xeon Phi 协处理器支持多种低功耗待机模式，但当使用 micsmc 测量功耗时，协处理器会立即结束当前的待机状态。因此，使用 micsmc 工具无法测量协处理器真实的待机功耗。

14.2.2 使用 micsmc 工具创建功耗和温度记录器

上一节介绍了一个监控协处理器功耗和 CPU 硅片温度的脚本示例。作为一个简单的功耗监控器或功耗计，在对 HPC 工作负载进行功耗、性能调整或平台热调整时这很有用。

然而，另一个值得具备的能力是将功耗和热数据记录到标准格式的文件中，如 .csv，以便使用其他功耗曲线或图更好地对原始功耗数据进行再处理。另外，用户也会需要在功耗、热记录数据上附加时间戳，与 HPC 应用程序的不同计算阶段进行时间匹配，以计算每瓦每秒的浮点运算次数。

为了处理更复杂的数据操作，用户可能需要使用比 bash 更高级的脚本语言。本节下一段代码示例将使用支持各种数据类型的 Python 语言，改进后的功耗和热记录脚本将输出如图 14-7 所示的记录文件。

图 14-7 的输出可以直接导入 Microsoft Excel 产生各种图表。因为功耗数据有了准确的时间戳，所以用户可将这些数据与 HPC 工作负载不同计算阶段的统计信息相匹配，以计算每瓦每秒的浮点运算次数。计算效率时，用户不应该将 HPC 代码初始化以及内存清理阶段的功耗数据包括在内，应该使用时间戳来匹配功耗数据和 HPC 程序的并行计算阶段。本例将选用高性能 LINPACK (HPL) 本机语言工作负载，使用 Intel MPSS RPM (micperfm RPM)，利用 micprun 应用来运行。图 14-8 显示了由本例功耗数据绘制的功耗曲线。

Time	Mic0 Power	Mic0 Temp
35:24.8	106	44
35:25.0	104	44
35:25.1	141	47
35:25.3	148	48
35:25.5	239	49
35:25.7	244	51
35:25.8	246	52

图 14-7 用 Python 语言编写的示例功耗和温度记录器输出

上图显示了在协处理器上运行 HPL 的整个功耗曲线。典型的 HPC 工作负载可能包括初始化、计算及内存清理阶段。为计算真正的每瓦每秒的浮点运算次数，用户需要将记录文件中不同的功耗抽样（在并行计算阶段获取）与 HPL 的并行计算阶段相匹配。有许多方法可以完成这一任务，比如，在 HPC 程序日志中计算阶段的开始和结束加时间戳。功耗数据需要进行过滤，以去除在程序并行计算阶段以外的数据，再对剩余的有效功耗抽样取平均值，即为并行阶段的平均功耗。大多数情况下，HPC 程序的计算阶段与功耗图中功耗最高的区间相关，因此可以通过这种方法找出计算阶段的开始和结束，而不需再与计算阶段的开始、结束时间相匹配。

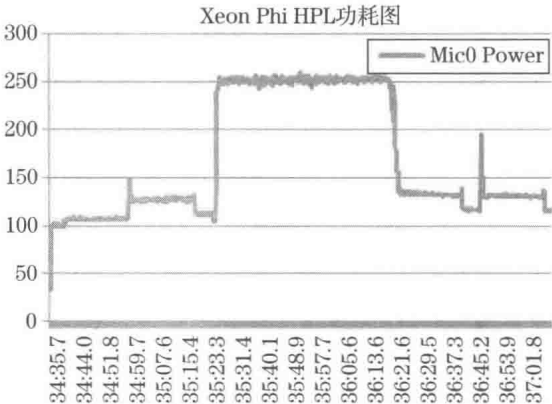


图 14-8 由功耗记录器脚本获取的 Xeon Phi HPL 功耗图

图 14-8 的功耗图可以清晰地显示稀疏矩阵上执行大量浮点计算的阶段，即使在原始的功耗记录文件中也能清晰地显示出来。

为分析 HPC 程序每瓦每秒的浮点运算次数，改进后的功耗、温度记录器将使用标准的 Python 模块，以及能够更灵活记录数据的程序结构。代码可从 lotsofcores.com 下载。

14.2.3 使用 IPMI 进行功耗分析

测量服务器或计算节点功耗的一个方便方法是使用常见的智能平台管理界面 (intelligent platform management interface, IPMI)。大多数 Intel 服务器都支持这一特征，从底板管理控制器获取服务器的总 AC 功耗。本节将使用开源的 IPMI 驱动程序和工具创建一个简单的计算节点 AC 功耗分析器。

首先需要确保服务器或计算节点配置了 IPMI 驱动程序和工具。一个简单的安装方法是使用 YUM 工具，命令如下：

```
$ yum install OpenIPMI OpenIPMI-tools
$ service ipmi start
```

要确定 IPMI 是否工作正常，可以输入下面的命令，该命令查询整个服务器或计算节点的传感器数据记录 (sensor data record, SDR)：

```
$ ipmitool sdr
```

图 14-9 显示了这条命令的输出，如图所示，用户可以通过 IPMI 访问很多传感器，比如主机 CPU 温度、风扇速度、电压以及总 AC 功耗等。不仅仅局限于服务器的功耗和热，而是整个服务器的健康状况，这在实际中非常实用。

输出中有太多传感器信息需要整理。而本节的功耗分析仅需要服务器总功耗（一些用户也可能需要主机服务器 CPU 热传感器数据）：

Pwr Unit Status	0x00	ok
Pwr Unit Redund	0x0a	ok
IPMI Watchdog	0x00	ok
Physical Scrtty	0x00	ok
FP NMI Diag Int	0x00	ok
SMI Timeout	0x00	ok
System Event Log	0x20	ok
System Event	0x00	ok
Button	0x00	ok
VR Watchdog	0x00	ok
Fan Redundancy	0x01	ok

图 14-9 典型的服务器传感器数据记录，突出显示的部分为功耗信息


```
$ ipmitool sensor get 'PS1 Input Power' 'PS2 Input Power'
```

如图 14-10 所示，该命令将只输出服务器上两个主要电源（PS1 和 PS2）的 AC 功耗。

SSB Therm Trip	0x00	ok
IO Mod Presence	0x00	ok
SAS Mod Presence	0x00	ok
BMC FW Health	0x00	ok
System Airflow	80 CFM	ok
BB P1 VR Temp	15 degrees C	ok
Front Panel Temp	17 degrees C	ok
SSB Temp	43 degrees C	ok
BB P2 VR Temp	16 degrees C	ok
BB Vtt 2 Temp	20 degrees C	ok
BB Vtt 1 Temp	17 degrees C	ok
HSBP 1 Temp	16 degrees C	ok
Exit Air Temp	24 degrees C	ok
LAN NIC Temp	33 degrees C	ok
System Fan 1	7987 RPM	ok
System Fan 2	7987 RPM	ok
System Fan 3	8085 RPM	ok
System Fan 4	7987 RPM	ok
System Fan 5	7987 RPM	ok
PS1 Status	0x01	ok
PS2 Status	0x09	ok
PS1 Input Power	332 Watts	ok
PS2 Input Power	0 Watts	ok
PS1 Curr Out %	40 unspecified	ok
PS2 Curr Out %	0 unspecified	ok
PS1 Temperature	19 degrees C	ok
PS2 Temperature	18 degrees C	ok
P1 Status	0x80	ok
P2 Status	0x80	ok
P1 Therm Margin	-80 degrees C	ok
P2 Therm Margin	-79 degrees C	ok
P1 Therm Ctrl %	0 unspecified	ok
P2 Therm Ctrl %	0 unspecified	ok
P1 ERR2	0x00	ok
P2 ERR2	0x00	ok
CATERR	0x00	ok
P1 MSID Mismatch	0x00	ok
CPU Missing	0x00	ok
P1 DTS Therm Mgn	-80 degrees C	ok
P2 DTS Therm Mgn	-78 degrees C	ok
DIMM Thrm Mrgn 1	-77 degrees C	ok
DIMM Thrm Mrgn 2	-73 degrees C	ok
DIMM Thrm Mrgn 3	-76 degrees C	ok
DIMM Thrm Mrgn 4	-74 degrees C	ok
Mem P1 Thrm Trip	0x00	ok
Mem P2 Thrm Trip	0x00	ok
Agg Therm Mgn 1	-36 degrees C	ok
BB +12.0V	11.83 Volts	ok
BB +5.0V	4.96 Volts	ok
BB +3.3V	3.24 Volts	ok
BB +5.0V STBY	4.94 Volts	ok
BB +3.3V AUX	3.24 Volts	ok
BB +1.05Vccp P1	0.83 Volts	ok
BB +1.05Vccp P2	0.85 Volts	ok
BB +1.5 P1MEM AB	1.50 Volts	ok
BB +1.5 P1MEM CD	1.50 Volts	ok
BB +1.5 P2MEM AB	1.50 Volts	ok

图 14-10 服务器输入功耗传感器以及功耗限制

BB +1.5 P2MEM CD	1.48 Volts	ok
BB +1.8V AUX	1.77 Volts	ok
BB +1.1V STBY	1.08 Volts	ok
BB +3.3V Vbat	3.21 Volts	ok
BB +3.3 RSR1 PGD	3.23 Volts	ok
BB +3.3 RSR2 PGD	3.03 Volts	ok
Sensor ID	PS1 Input Power (0x54)	
Entity ID	10.1	
Sensor Type (Analog)	Other	
Sensor Reading	332 (+/- 0) Watts	
Status	ok	
Upper Non-Critical	868.000	
Upper Critical	920.000	
Sensor ID	PS2 Input Power (0x55)	
Entity ID	10.2	
Sensor Type (Analog)	Other	
Sensor Reading	0 (+/- 0) Watts	
Status	ok	
Upper Non-Critical	868.000	
Upper Critical	920.000	

图 14-10 （续）

读者可能注意到了其中一个目标服务器中电源的功耗为 0W。这个输出很正常，因为这是一个双电源服务器，在当前工作负载条件下，仅需要一个电源为服务器供电。

在 HPC 应用程序的计算阶段，用户可以使用简单的脚本发射 IPMI 命令查询电源功耗，并加上时间戳，将这些输出信息记录到文件中，即可通过后期处理计算出程序运行阶段的平均功耗。前面提到的协处理器功耗和热监控脚本也可用于采集服务器或计算节点的总 AC 功耗。

14.3 基于硬件的功耗分析方法

本章已经通过示例展示了如何使用 Intel MPSS 工具以及如 IPMI 这类工业界标准工具来创建简单的功耗分析器，分析和估算每瓦每秒的浮点运算次数。如果编写 HPC 程序的软件开发者想查看系统工作的能效，这些工具会非常有用。

然而，若要以非侵入式的方式测量功耗（即在被测量设备上没有额外的软件负载），用户需要使用基于硬件的功耗分析方法。有几种不同的方法可以完成这项工作，但本节将介绍 Green 500 采用的其中几种测量方法，以及基于硬件功耗分析中一些特殊任务的解决方法。

测量 HPC 集群总功耗的方法之一是使用电源分配单元（power distribution unit, PDU）。这些设备的外形类似 AC 电源插排，内嵌有功耗测量功能，在测量集群功耗中很有用。在实际使用中，很多 PDU 内嵌有 Web 服务器，管理员或用户可以远程监控在 HPC 集群负载下的瞬时功耗。这些智能的 PDU 可以与简单网络管理协议（simple network management protocol, SNMP）及 Python 脚本一起使用，以创建集群级别的功耗记录器。使用 PDU 作为功耗分析器的优势之一是所有机架上的设备功耗都将被获取，包括结构（交换器），否则没有简单的方法能获取结构的 AC 功耗。数据中心机架上所有硬件设备一般都从 PDU 获取电能，所以用户可以及时通过智能 PDU 获取任意时刻的功耗。图 14-11 是一个典型的 PDU。

另一种计算节点级基于硬件的功耗分析方法是使用 AC 功耗分析器。这些设备连接在 AC 电源插槽和负载（服务器和计算节点）之间。这些 AC 功耗分析器也可以非侵入式地测量功耗，不会影响被测量系统上的任何软件线程，用户不会希望计算节点使用计算资源记录

自身的功耗数据。通过使用非侵入式测量方法,用户可将测量和记录的工作运行在其他系统上,该系统通过发出 SNMP 和 GPIB 命令获取和记录功耗数据以供后期处理。

基于硬件的协处理器功耗分析方法

本章最后一个内容是使用基于硬件的协处理器功耗分析器对被测试的 HPC 系统非侵入式地执行高级分析。该方法使用 PCIe 插入器和测量电阻,以及 National Instruments LabVIEW Signal Express 软件包来实时抽样和执行协处理器功耗分析。

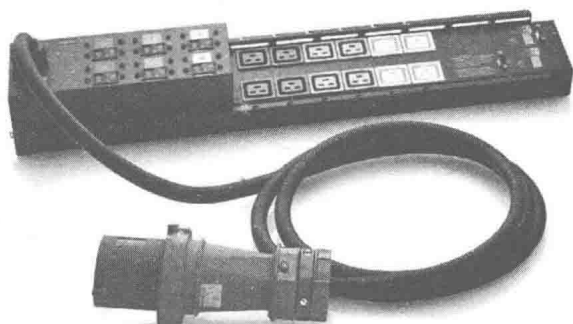


图 14-11 一个典型的 PDU

图 14-12 展示了本章使用的基于硬件的功耗分析器,该分析器使用标准的、现成的组件 (PCIe 插入器使用 Adex Electronics 公司的产品) 以及 National Instruments LabVIEW 软件。这种配置可以非常高的频率抽样功耗数据 (每秒 25000 个采样值),精度也很高 (24 位 ADC)。这种方法基于 PCIe 插入器读取 PCIe Express 的功耗 (PCIe 3.3 V 和 PCIe 12 V) 以及外部 2×3 和 2×4 12V 补充电源轨的测量电阻。用户只需简单地将原始电压、电流信号输入给数据采集器单元,LabVIEW 即可进行抽样来计算协处理器的瞬时功耗。图 14-13 显示了用 LabVIEW 进行测量并记录功耗的示例。

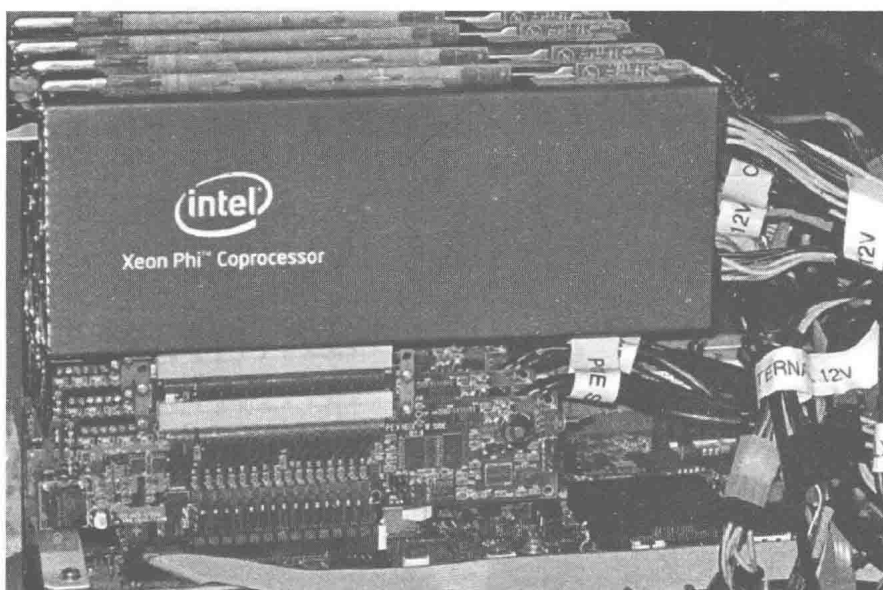


图 14-12 一个 Xeon Phi 协处理器硬件功耗分析器

图 14-13 的功耗截图反映了当增加一个 HPC 工作负载所使用的协处理器内核数量时,功耗增长与活跃内核数增加有怎样的线性关系。图中的水平轴表示时间,竖直轴表示以瓦为单位的功耗,原始的功耗数据以 5 kHz 频率抽样,使用 LabVIEW 实时计算 1 秒移动平均值 (使用软件中的有限脉冲响应滤波器),在 HPC 负载运行过程中产生了这幅图。这种方法的 最大优势是用户在执行高级的功耗分析时不必要求处理器或协处理器进行计算或数据记录。所有的硬件线程均可用来运行 HPC 工作负载。

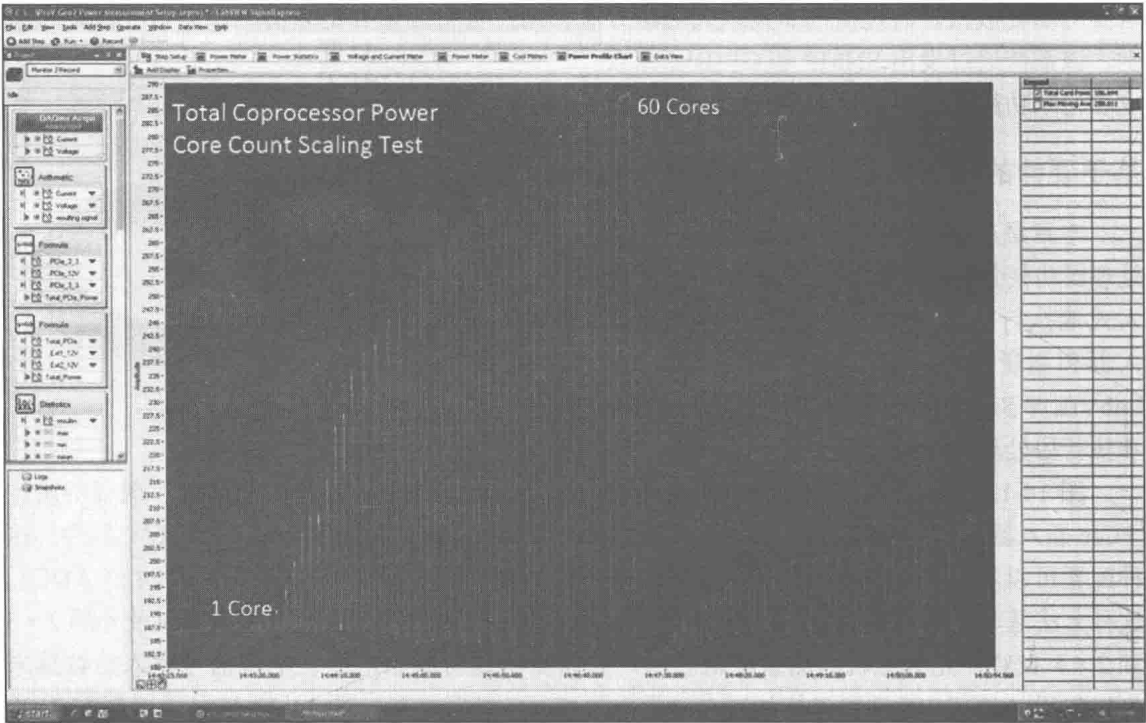


图 14-13 使用 LabVIEW 测量的协处理器总功耗

用户在统计功耗时需要考虑全部 4 个电源轨输入 Intel Xeon Phi 协处理器的功耗（一共 300 W TDP），如图 14-14 所示。

PCIe 3.3 V 和 PCIe 12 V 可以通过 PCIe 插入器进行抽样，2×3 和 2×4 补充电源可以通过将标准测量电阻连接到协处理器输入电源进行采样。本例中 PCIe 插入器使用 10 mΩ 检测电阻，2×3 和 2×4 的补充 12V 电源使用耐用的 25 mΩ 测量电阻。这 4 个电源轨的原始功耗数据如图 14-15 所示。

协处理器电源轨(V)	电源轨规格(W)
PCI Express 3.3	10
PCI Express 12	65
2×3 补充电源轨	75
2×4 补充电源轨	150

图 14-14 协处理器所有输入电源轨

图 14-15 展示了协处理器实时瞬时功耗，每秒有数千次抽样。图中将 PCIe 3.3V 和 PCIe 12V 合并为一个总的 PCIe 功耗，将此功耗与 2×3 和 2×4 的补充 12V 电源相加得到截图上部所示的总功耗。这些数据也将被 LabVIEW 保存下来以备对原始数据进行后期处理。LabVIEW 也可给出文本格式的功耗数据统计信息，如图 14-16 所示。

图 14-16 展示了 LabVIEW 能够给用户提供的统计信息，包括最小值、最大值和平均值。需要注意的是，PCIe 3.3V 辅助电源显示的是无实际功耗，因为协处理器并不会从 PCIe 3.3V 辅助电源获取电能，PCIe 3.3V 是为网卡预留的，为网卡提供待机供电以便有事件时被唤醒。

本节使用了上文介绍的基于硬件的功耗分析器来确保本章开头提到的基于软件的功耗分析器的置信区间在可接受的范围内。做这种相关性测试需要确保计算总功耗的方法相同。Intel MPSS micsmc 工具采集板上 SMC 的功耗数据，并在内部用 1 秒移动平均法计算出功耗。本章在 LabVIEW Signal Express 中也采用 1 秒移动平均计算测量区间的功耗，以便进行对等的比较。在本章的配置中，基于硬件的功耗分析器通常得到稍高的总卡功耗（4～6 W），

这是因为 SMC 不监控 PCIe 3.3V 电源轨，只监控 12V 电源轨（PCIe 12V，2×3 和 2×4）。LabVIEW Signal Express 的数据记录特征可以通过未加工的文本或以科学计数法表示的功耗数据提供给用户非常精确的原始功耗数据，以备进一步的后期处理。

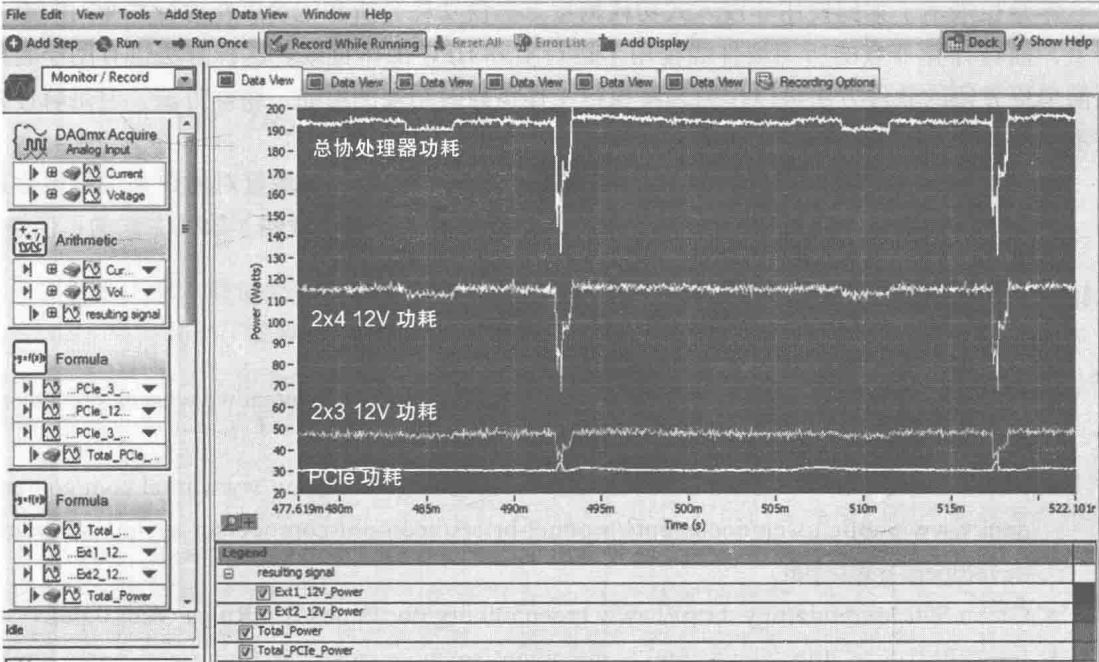


图 14-15 协处理器原始瞬时功耗

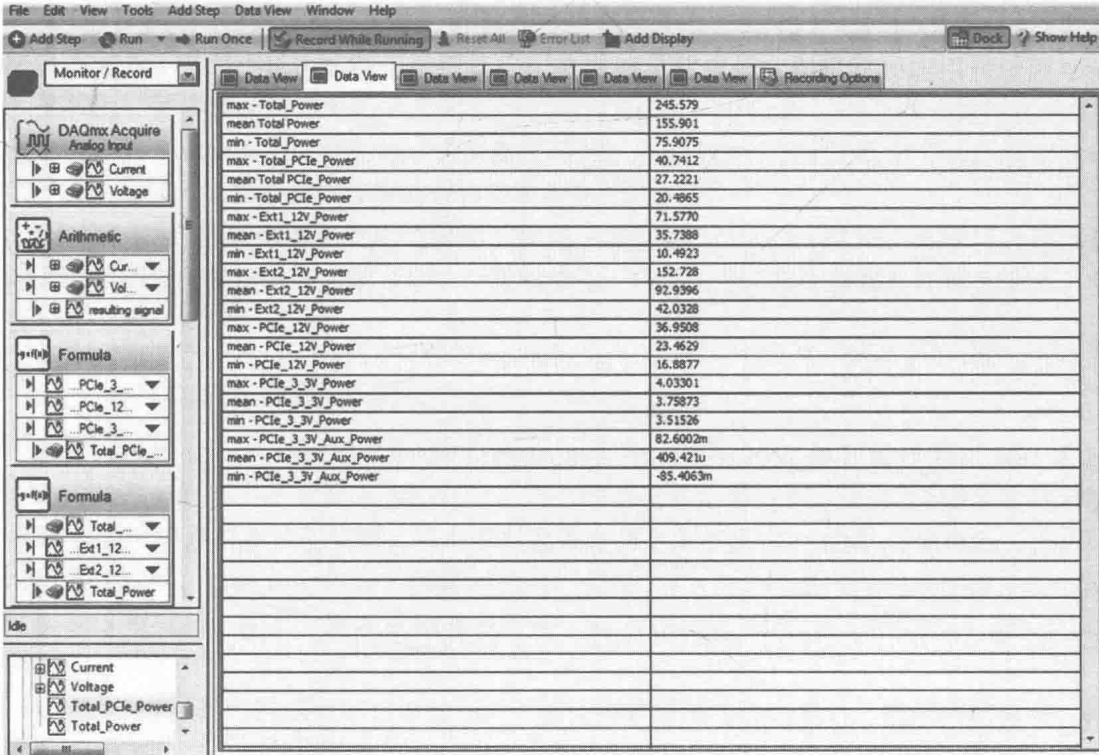


图 14-16 协处理器上实时电源轨统计信息

14.4 总结

因为功耗已经成为非常重要的制约因素，所以本章讨论了使用软件工具及简单脚本来测量和分析协处理器功耗的几种关键方法。

本章也展示了如何使用非侵入式功耗测量方法使功耗分析和记录工作运行在专门的服务器上，而将计算节点的所用硬件线程用于运行实际 HPC 工作负载。这种方法也有助于验证更简单更方便的软件方法可以作为理解执行工作负载时功耗的可靠、精确方法，但是软件方法不能测量待机功耗。

能够使用软件方法对运行中的 HPC 系统进行准确的功耗和热测量对用户来说是很幸运的。根据本章在 Intel 实验室进行的基于硬件的功耗分析，基本上证明了软件方法的可信性。

14.5 更多信息

下面是本章推荐的一些额外阅读材料：

- Intel Xeon Phi™ coprocessor: Datasheet. <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html>.
- Intel® Xeon Phi™ System Software Developer's Guide. <http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf>.
- Green 500 Methodology. http://www.green500.org/docs/pubs/RunRules_Ver1.0.pdf.
- Intel IPMI Site. <http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html>.
- 本章及其他章代码下载地址 <http://lotsofcores.com>。

集成 Intel Xeon Phi 协处理器至集群环境

Paul Peltz*, Troy Baer*, Ryan Braby*, Vince Betro*, Glenn Brook*, Karl Schulz†

* 美国, 田纳西大学诺克斯维尔分校、NICS; † 美国, Intel 公司

15.1 早期探索

田纳西大学和橡树岭国家实验室共同管理着联合计算科学研究所 (JICS)。2007 年 JICS 的 Kraken 项目获得了国家科学基金 (NSF) 奖, 并因此建立了计算科学国家研究院 (NICS)。2011 年年初, NICS 和 Intel 达成了多年的战略合作关系, 以寻求基于 Intel 集成众核架构 (Intel MIC, 目前正式更名为 Intel Xeon Phi 产品系列) 的下一代高性能计算 (HPC) 解决方案。为了回报 NICS 为 Intel 所提供的应用测试、性能测试结果和专家反馈, Intel 让 NICS 提前接触了 Intel MIC 技术。由此产生的合作关系, 让 JICS 内部的卓越应用程序加速中心可以在未发行的产品 Knights Ferry (KNF) 的软件开发平台上进行软件性能测试和开发。把平台连接在一起后形成了 Intel 外部首个配备有 Intel MIC 技术的集群。此后不久, NICS、Intel 与 Cray 公司一起部署一个双节点的类似“封装集群”的 Cray CX1 超级计算机, 其中每个节点上配有一个 KNF 协处理器。之后, 又与计算机系统生产商 Appro 一起部署了包含 4 个节点的常规集群, 其中每个节点配备两个 KNF 协处理器。

在 2011 年的高性能计算、网络、存储和分析国际年会 (SC11) 上, NICS 和 Intel 宣布战略合作关系, 并且 NICS 发布了其科技应用方面的早期成果, 即将科技应用移植到 Intel MIC 架构上。其中涉及的主要科学应用代码包括 NWChem 和 Enzo, 以及电子结构代码 Elk 和一个内部的 Boltzmann 计算流体动力学解算器等。在报告中, NICS 给出了这些应用的移植状态信息及部分性能数据。报告的公告和视频见 15.12 节。

JICS 在 KNF 协处理器上完成的前期工作进一步扩展为 Beacon 项目, 该项目目前由美国国家科学基金会和田纳西大学资助, 用于探索 Intel Xeon Phi 协处理器在计算科学和工程应用上的效果。

15.2 Beacon 系统的历史

最初的战略合作伙伴关系扩展到了 Beacon 项目, 并在 2012 年开始成为由 NSF 资助 (编号 1137097) 的网络基础设施战略技术。提出的系统是由通过单个 FDR 无限带宽技术转换器互连的 16 节点所组成的集群, 每个节点配置有两个未产品化的 KNF 协处理器。这是一个比较小的集群, 预期会非常易于部署和管理。但是, KNF 卡的操作系统软件尚未计划用于无盘 NFS 根安装。因此, 需要选用其他方法, 而不能采用 NICS 提出的基于无盘 NFS 根的集群管理方法。

早期用户 (包括田纳西大学以外的用户) 将代码移植至协处理器后, 能够获得显著的性能提升。Beacon 项目也包括了从 KNF 到代号为 Knights Corner (KNC) 的协处理器产品的升级。Beacon 项目非常成功, 以至于田纳西大学决定给出超出原计划投资数目的资金, 使

得 Beacon 项目可以部署更大的和更高功耗效率的集群。

这种新的、升级的 Beacon 超级计算机迅速部署完成。它包含 48 个 Cray CS300-AC 节点 (Appro 公司的 Greenblade 节点, 随后, Appro 被 Cray 收购), 每个节点有两个 Intel Xeon E5 处理器和 4 个 Intel Xeon Phi 协处理器。来自 Intel、Appro 和 NICS 等世界各地的团队参与到这个项目中。项目的主要任务是完成集群构建、系统调整, 并且及时执行高性能测试软件包的 Linpack, 以参与 Green500 的评选。最后, Beacon 获得了 SC12 Green500 的第一名。会议结束后, 参与了这个项目的 Intel 研究员 Pradeep Dubey 说: “这是我在 Intel 的工作中最卓越的一次合作经历”(见图 15-1)。

美国 NSF 的研究人员正在用 Beacon 来优化应用程序, 使它们能够充分利用 Intel Xeon Phi 协处理器和未来高度并行架构的高度并行能力。请参阅 15.12 节, 以获取更多关于 Green500 和 Pradeep Dubey 博客的信息。

15.3 Beacon 系统的架构

15.3.1 硬件环境

Beacon 是一个 Cray CS300-AC 集群, 由通过 FDR 无限带宽技术互连的 48 个计算节点和 6 个 I/O 节点组成。每个计算节点包含两个 8 核 Intel Xeon E5-2670 处理器, 256GB 内存, 960GB SSD 存储, 以及 4 个 Intel XeonPhi 5110P 协处理器。总体而言, Beacon 总共有 768 个常规内核, 11520 个协处理器, 12TB 系统存储, 1.5TB 协处理器存储, 以及超过 73TB 的 SSD 存储。其中, 11520 个协处理器提供了超过 210TFLOPS 的计算能力。Beacon 支持两种以太网网络: 一种是专用内部网, 另一种是用于外部系统管理的网络。

15.3.2 软件环境

在撰写本书时, Beacon 上部署的 Intel 众核平台软件栈 (Intel MPSS) 的版本是 3.2.3。Intel 提供的 Intel MPSS 文档和相关的工具提供了在工作站和 “vanilla” 集群环境下的安装支持。但是在具有广泛用途的 HPC 集群 (如 Beacon) 下, 扩展和个性化定制协处理器的安装仍然存在很多独特的挑战。本章重点介绍了在已有集群上完成像集成这样的关键步骤的方法。需要注意的是, 本章作者假设读者已经熟悉 Intel 提供的 readme 和用户指南, 因为作者将引用其中的参考工具和配置。Intel MPSS 的下载和参考指南链接见 15.12 节。

作者认为有几个系统管理工具对于正确地部署和维护集群非常必要。其中一个最重要的工具是配置管理器, 如 Puppet、Chef、Salt 或者 Ansible。这几个工具都很好用, NICS 选择使用 Puppet。使用配置管理器有很多好处, 其中最主要的原因是在调用 Intel MPSS 的 micctrl 实用程序后恢复配置文件。配置管理器也可以用于将 Intel MPSS 配置文件分发到集群中所有的计算节点。作者还建议安装集群资源和工作负载管理器, 以管理用户的作业调度。大多数通用集群会使用一个作业调度工具, 而 Beacon 则同时使用 TORQUE 和 Moab。对于每个单独作业的 Intel Xeon Phi 协处理器环境的管理部署, TORQUE 或者类似的资源管理器很重要。15.5 节会对这部分内容进行更加详细讨论。用一个工具来管理用户环境也非常必要, 可以用 modules 或 Lmod。作者在本章中提到的最后一个工具是 pdsh, 这是一个多线程远程 shell 客户端, 它可以在远程主机上并行地执行命令。本节提到的工具的相关链接参见 15.12 节。



图 15-1 Green500 奖

15.4 Intel MPSS 安装步骤

Beacon 的操作系统安装是有盘的 (diskful), 所以本章中 Intel MPSS 栈的安装信息也基于这类环境。有盘安装是指节点上的操作系统是安装在节点的物理磁盘驱动上的。这对系统管理员而言无疑是一个挑战, 因为可能出现节点间的配置和软件程序包不同步的情况。例如, 如果一个节点必须脱机进行硬件调整, 而此时其他节点恰巧需要更改配置或者软件程序包。如果这次更改不是通过配置软件而是手动完成的, 那么节点间的配置可能会略有不同。无盘的 NFS 根文件系统可以解决这个问题。在每次节点重新启动时都从单个系统映像启动, 这样所有节点具有完全相同的操作系统映像。Beacon 最初开展了只读 NFS 根文件系统映像在节点上的部署, 但是由于时间限制和 Intel MPSS 早期版本的技术限制, 我们没有来得及在 Beacon 上实现无盘根。计算节点的 IP 地址也设置为静态的, 以简化协处理器配置文件的管理。

15.4.1 系统准备

《MPSS 3.2.3 用户指南》的第 19 节为如何在集群上安装 Intel MPSS 栈提供了一些建议, 但未提及对于 HPC 集群存在的一些特定问题。可以在 15.12 节中获取 Intel MPSS 软件和文档的链接。在使用本章中提供的脚本和工具前, 需要在集群做一些准备工作。如果读者在集群上使用的是动态主机配置协议, 那么需要进行静态 IP 地址配置, 而不能使用动态 IP 地址池, 以确保每个主机和协处理器都绑定了固定的 IP 地址。下面所示的结构化的 /etc/hosts 文件对于 Intel MPI 库和脚本的运行非常必要。在本章假设性的例子中, 我们选择私有子网中一段连续的 IP 地址范围, 其中一个 IP 地址对应一个主机的 eth0 接口, 另外两个的 IP 地址映射到安装在主机上的两个协处理器上。在 Beacon 中, 每个计算节点上有 4 个 IP 地址分别关联到 4 个协处理器上。请注意, 如果需要, 这些脚本很容易修改, 以引用不同的文件, 而不引用 /etc/hosts。但是, 在协处理器上存放一样的 /etc/hosts 文件也是可取的。如果 hosts 文件放置在 /var/mpss/common/etc/ 目录下, 那么协处理器将在协处理器的文

件系统中引导，并用这个文件覆盖 `/etc/hosts` 文件。请参阅《MPSS 用户指南》，了解协处理器的 `/var/mpss` 配置目录如何工作（见图 15-2）的更多详细信息。

10.0.0.12	node001	node001-eth0
10.0.0.13	node001-mic0	
10.0.0.14	node001-mic1	
10.0.0.15	node002	node002-eth0
10.0.0.16	node002-mic0	
10.0.0.17	node002-mic1	

图 15-2 `/etc/hosts` 示例文件

直接 `ssh` 访问该协处理器需要 `ssh` 密钥。请参阅《MPSS 用户指南之 RSA/DSA 密钥生成》的第 19.2 节以了解如何生成这些文件，以便将它们分发到各个协处理器。

如果集群使用 Infiniband 架构以达到高速互连，那么需要在集群上安装 OpenFabrics 企业分发（OFED）软件栈。在撰写本书时，Intel 提供了两个 OFED 可用版本：1.5.4.1 和 3.5-1-MIC。版本 1.5.4.1 是于 2012 年 2 月发布的，所以它是 OFED 非常老的一个分支。这也是 Intel-MPSS 栈从首次公开发行人以来一直支持的版本。虽然 Intel 在 MPSS 栈中仍为 1.5.4.1 版本提供驱动程序和库支持，但是 Intel MPSS 栈的 3.2 版本添加了对新一代 3.5-1-MIC 版本的支持。这是官方的 OFED 3.5 版本的分支，其中包含了对 Intel MIC 架构的主机端扩展支持。根据所安装的版本，会有不同的安装说明。从《MPSS 用户指南》的第 2 节可以获取更多信息。

注意：对于版本 1.5.4.1，需要安装 OFED 栈和 `$MPSS_LOCATION/ofed/` 目录下的 RPM。如果安装了新的 3.5-1-MIC 版本，则仅需要安装 OFED 栈，`$MPSS_STACK/ofed/` 目录下的 RPM 文件不用再安装。

15.4.2 安装 Intel MPSS 栈

在 Beacon 系统上安装 Intel MPSS 栈时，需要先下载并解包 Intel MPSS 文件到一个共享的目录，如 NFS 共享目录，这样它就可以被所有计算节点所访问。在撰写本书时，3.2.3 是最新的可用版本，因此本章的安装和配置说明也将基于这个版本，但之后的 3.X 版本应该都是类似的。默认情况下，Intel MPSS 假设你正在使用的是 Red Hat 公司的 Linux 发行版（RHEL）或者 CentOS 任意版本的标准内核。例如，如果从网上下载 RHEL/CentOS-6.2 版本的 Intel MPSS 3.2.3 包，那么它的 Intel MPSS 驱动程序会与 RHEL/CentOS 6.2 之前版本的内核不相容。但是新的 Intel MPSS 会将所有 OS 发布版本打包到一起。在 RHEL/CentOS 6.2 内核中的内核版本是 2.6.32-220。这一版本内核中存在安全漏洞，管理员需要使用补丁来保证集群安全。Intel MPSS 为主机驱动（和 OFED 内核模块）提供了用于针对新的内核版本进行重建的 RPM。下面重点介绍的示例包含了从源代码（见图 15-3）构建主机协处理器驱动程序步骤。

- 1) Ensure the prerequisites are installed:
> `sudo yum install kernel-headers kernel-devel`
- 2) Regenerate the Intel(R) MPSS driver module package:
> `cd $MPSS_LOCATION/src/`
> `rpmbuild --rebuild \ mpss-modules-*.el6.src.rpm`
- 3) Copy the mpss-modules binary RPM from the `$HOME/rpmbuild/RPMS/x86_64` directory to `$MPSS_LOCATION` (overwriting the mpss-modules binary RPM that was provided with the precompiled driver).

图 15-3 内核重建指令

准备工作结束后，通过图 15-4 中的指令，使用 pdsh 工具为所有计算节点安装 Intel MPSS 包。

```
sudo pdsh -w cluster\[xxx-xxx\] yum -y install \
--nogpgcheck --nogpgcheck --nogpgcheck --nogpgcheck \
--nogpgcheck --nogpgcheck --nogpgcheck --nogpgcheck \
$MPSS_LOCATION/*.rpm
```

图 15-4 Intel MPSS 安装指令

另外，还可以直接使用 yum 安装。根据所采用的基础配置管理策略，Intel MPSS 的 RPM 也可以集成到一个计算设备并在配置管理系统的支持下直接分发。

为了支持群集上分布式协处理器之间的“对称”MPI 通信，需要安装相关的 Intel MPSS OFED 包。如果 OFED-1.5.4.1 包已在计算节点上安装了，也有必要通过 pdsh 或配置管理器安装 Intel MPSS OFED RPM。注意，和主机的 Intel MPSS 协处理器驱动程序类似，如果使用该 Intel MPSS 版本不支持的内核，则 OFED 内核模块将需要重建。重建过程可以通过 \$MPSS_LOCATION/src/ 目录下的命令（见图 15-5）完成。

```
rpmbuild --rebuild ofed-driver-*.src.rpm
```

图 15-5 OFED RPM 重建指令

RPM 可以从 \$HOME/rpmbuild/RPMS/x86_64/ 目录下复制到 \$MPSS_LOCATION/ofed/ 目录下。旧的 OFED 内核模块的 RPM 应该移至其他目录，避免两个版本同时安装。就像前面提到的，如果使用 OFED 3.5 系列则不需要重建 OFED 驱动程序。

最后一步是检查是否需要更新 Intel Xeon Phi 协处理器的固件。MPSS 用户指南描述了使用提供的 micflash 包更新固件的过程。

15.4.3 生成和定制配置文件

Intel 提供的 micctrl 命令可以完成很多事情，以创建配置文件和管理协处理器。正如前面所提到的，此命令提供的默认配置可用于快速完成一台独立主机或工作站环境的设置。对于群集环境，经常需要将 micctrl、本地基于脚本的定制和配置管理系统相结合，来完成集成和部署过程。下面的内容描述了在 Beacon 上实现的特定方法。

如果读者是首次安装或正在使用一个新的 Intel MPSS 版本，那么在开始的时候，需要通过图 15-6 中所提供的指令在特定的计算节点上生成 Intel MPSS 配置文件。

```
sudo micctrl --cleanconfig
sudo micctrl --initdefaults
```

图 15-6 micctrl 命令

Intel MPSS 3.2.3 中的 micctrl 实用程序将自动扫描本地主机上的所有可用用户，以建立协处理器 ssh 密钥访问。然而，为了与资源管理器相协调，许多 HPC 站点需要管理协处理器的 ssh 访问。使用图 15-7 中给出的附加命令，可以将默认的配置改为只提供最少数目的协处理器管理员用户。

```
sudo micctrl --userupdate=none
```

图 15-7 micctrl 命令

由上述命令生成的文件将用作每个节点的模板——不同节点的 MPSS 配置文件在网络配置上会稍有不同。首先，将文件 /etc/mpss/default.conf 和 /etc/mpss/mic0.conf

复制到一个位置，如 /ect。在这个目录下运行图 15-8 中提供的 mic-create-conf.sh 脚本，以产生特定主机配置文件。

```
#!/bin/bash
### Script to create /etc/mpss/mic*.conf
### Currently setup for MPSS 3.2

if [ -d /sys/class/mic ]
then
  for MICN in $( cd /sys/class/mic ; echo mic* )
  do
    MICIPADDR=$(grep ${HOSTNAME}-${MICN} /etc/hosts | awk
    '{print $1}')
    HOSTIPADDR=$(grep ${HOSTNAME}-eth0 /etc/hosts | awk
    '{print $1}')
    if [ -n "$MICN" -a -n "$MICIPADDR" -a -n
    "$HOSTIPADDR" ]
    then
      # generate /etc/mpss/default.conf files
      cat /etc/default.conf.template | sed
      "s/HOSTIPADDR/$HOSTIPADDR/g" > /tmp/default.conf
      delta=$(diff -u /tmp/default.conf
      /etc/mpss/default.conf | wc -l)
      if [ -n "$delta" -a $delta -gt 0 ]
      then
        cp /etc/mpss/default.conf /tmp/default.conf.bak
        cp /tmp/default.conf /etc/mpss/default.conf
      fi
      # generate /etc/mpss/micN.conf files
      cat /etc/micN.conf.template | sed
      "s/NODE/$HOSTNAME/g" | sed "s/HOSTIPADDR/$HOSTIPADDR/g" |
      sed "s/MICN/$MICN/g" | sed "s/MICIPADDR/$MICIPADDR/g" >
      /tmp/${MICN}.conf
      delta=$(diff -u /tmp/${MICN}.conf
      /etc/mpss/${MICN}.conf | wc -l)
      if [ -n "$delta" -a $delta -gt 0 ]
      then
        cp /etc/mpss/${MICN}.conf /tmp/${MICN}.conf.bak
        cp /tmp/${MICN}.conf /etc/mpss/${MICN}.conf
      fi
    fi
  done
fi
```

图 15-8 mic-create-conf.sh 脚本

在集群环境下，可以将协处理器作为网络拓扑的一部分并且通过 TCP/IP 完成协处理器与协处理器间的直接通信。这可以通过外桥接来实现。可以在图 15-9 所示的示例模板文件 default.conf 中查看 Bridge 行了解相关语法（文件中的 HOSTIPADDR 变量需要替换为特定节点的 IP 地址）。

在图 15-10 所示 micN.conf.template 文件中的 Network 行中添加一个选项是非常有必要的。

mic-create-conf.sh 脚本假定存在 /etc/hosts 文件，且该文件与图 15-2 中的语法一致。根据需要定制了配置文件模板和脚本后，执行 mic-create-conf.sh 脚本来生成配置文件。同时，也需要有定制的 ifcfg-ethX 和 ifcfg-micbr0 文件来使外部桥接正常工作。这些文件位于 /etc/sysconfig/network-scripts/。ifcfg 示例文件如图 15-11 和图 15-12 所示。

在这种情况下，eth0 是内部私有 TCP/IP 以太网网络。根据需要要用相应的接口替换 eth0。ifcfg-micbr0 文件调用 grep 指令，以从 /etc/hosts 文件中获得相应的 IP 地址。这种方法使得同一个 ifcfg-micbr0 文件可以在整个集群上使用。协处理器还需要有

如图 15-13 所示的 ifcfg-micX 文件。

```
# Source for base of embedded Linux file system
Base CPIO /usr/share/mpss/boot/initramfs-
knightscorner.cpio.gz

# MIC card unique overlay files such as etc, etc.
CommonDir /var/mpss/common

# Additional command line parameters. Caution should be
used in changing these.
ExtraCommandLine "highres=off"

# MIC Console
Console "hvc0"

ShutdownTimeout 300

# Storage location and size for MIC kernel crash dumps
CrashDump /var/crash/mic/ 16

Bridge micbr0 External HOSTIPADDR 24 9000

PowerManagement "cpufreq_on;corec6_off;pc3_off;pc6_off"
```

图 15-9 default.conf 模板文件

```
Version 1 1

# Include configuration common to all MIC cards
Include default.conf

# Include all additional functionality configuration
files by default
Include "conf.d/*.conf"

# Unique per card files for embedded Linux file system

MicDir /var/mpss/MICN

# Hostname to assign to MIC card
Hostname "NODE-MICN"

# MAC address configuration
MacAddrs "Serial"

Network class=StaticBridge bridge=micbr0 micip=MICIPADDR
mtu=64512 modhost=no modcard=no

# MIC OS Verbose messages to console
VerboseLogging Disabled

# MIC OS image
OSimage /usr/share/mpss/boot/bzImage-knightscorner
/usr/share/mpss/boot/System.map-knightscorner

# Boot MIC card when MPSS stack is started
BootOnStart Enabled

# Root device for MIC card
RootDevice ramfs /var/mpss/MICN.image.gz

Cgroup memory=disabled
```

图 15-10 micN.conf.template

对于系统中的每个协处理器，都要创建一个 ifcfg-micX 文件。模板文件、ifcfg 文件和脚本需要能够被配置管理器所访问，以在整个集群下进行部署。把配置文件发布到整个集群后，需要运行 mic-create-conf.sh 脚本，然后通过 pdsh 执行图 15-14 中的指令。


```
DEVICE=eth0
BOOTPROTO=static
IPV6INIT=no
MTU=9000
NM_CONTROLLED=no
ONBOOT=yes
TYPE=Ethernet
BRIDGE=micbr0
USERCTL=no
```

图 15-11 ifcfg-eth0 文件

```
DEVICE=micbr0
TYPE=Bridge
BOOTPROTO=static
IPV6INIT=no
MTU=9000
NM_CONTROLLED=no
ONBOOT=yes
TYPE=Ethernet
IPADDR=$(grep `hostname -s`-eth0 /etc/hosts | \
awk '{print $1}')
```

图 15-12 ifcfg-micbr0 文件

```
DEVICE=mic0
ONBOOT=yes
NM_CONTROLLED="no"
BRIDGE=micbr0
```

图 15-13 ifcfg-mic0 文件

```
sudo micctrl --resetconfig
```

图 15-14 更新协处理器的 micctrl 命令

```
sudo micctrl --initdefaults
```

图 15-15 ifcfg-mic0

micctrl 命令能够更新协处理器上的配置文件，使其与在主机上生成的配置文件一致。使用 chkconfig 命令使 mpss 和 ofed-mic 服务从所需的运行级别开始启动。此时整个集群就可以重启了，每个主机和协处理器在内部私有网络中都应该可以访问到。

15.4.4 MPSS 升级

在撰写本书时，没有直接的 Intel MPSS 版本升级方法。为了更新栈，需要删除之前安装的版本，然后安装新的版本。比较好的做法是在部署 Intel MPSS 的新版本之前，先在单个计算节点上进行测试安装和配置。从作者的经验来看，这个过程从来都不只是简简单单删除之前栈，然后安装新版本，主要是因为需要将新选项增加到配置文件中，而且语法也可能会有变化。我们希望在之后的版本中，这种配置文件版本间的变化将不那么频繁。

Intel MPSS 提供了 uninstall.sh 脚本，用于删除当前运行的栈。在测试节点上，关闭配置管理器，安装新的 MPSS 栈，然后运行图 15-15 所给出的指令。

对比新旧配置文件的语法，查看两者的差别。必要时调整模板文件并重新启动待测节点，以测试配置文件的有效性。如果一切工作正常，则更新配置管理器中的配置文件。然后可以在测试节点上重启配置管理器。

检查是否有 15.4.2 节提到的固件更新，必要时更新协处理器。一旦所有这一切都已经完成，使用 pdsh 删除 Intel MPSS 并在每个计算节点上安装新版本。

15.5 建立资源和工作负载管理器

15.5.1 TORQUE

资源管理器是管理协处理器最重要的应用。本章介绍的例子是 TORQUE，但涉及的概念很容易转移到其他资源管理器，如 SLURM。TORQUE 序言程序的主要责任是设置用户环境，安装外部文件系统，并检查协处理器是否“健康”。尾声程序将负责删除临时文件，终止垃圾进程，卸载文件系统，并使节点返回一个干净的状态，为下一份作业做准备。从 <http://lotsofcores.com> 上可以找到更多与 TORQUE 序言程序和尾声程序相关的例子。

15.5.2 序言程序

建议为用户建立一个临时目录，用来存储之前生成的 ssh 密钥、用户数据，以及 Intel MPI 库运行所必需的一些 Intel MPI 二进制文件（见图 15-16）。

```
tmp=/tmp/pbstmp.$jobid
mkdir -m 700 $tmp && chown $user.$group $tmp
cp ssh-id_rsa.key $tmp/.micssh/
```

图 15-16 TORQUE 序言程序中的 \$tmp 变量

在协处理器上创建用户环境时，可以通过创建一个基于用户所加载模块的 .profile 来实现。这个例子中引用的环境变量是部署在 Beacon 上的模块所提供的，但是这些变量也可以静态设置。\$IMPI 是 Intel-MPI 的安装位置，\$INTEL_COMPILERS 是 Intel 编译器的安装位置（见图 15-17）。

```
cat <<EOF > $tmp/profile
export TMPDIR=$tmp
export PATH=$TMPDIR/bin
export LD_LIBRARY_PATH=$TMPDIR/lib:$IMPI:$INTEL_COMPILERS
EOF
```

图 15-17 在 TORQUE 序言程序中创建 .profile

Intel Xeon Phi 协处理器有时没有被正确地初始化，或者由于之前的使用而处于糟糕的状态。因此，建议在使用它们之前，做一些基本的“健康”检查。如果检查失败，序言程序会尝试重新启动协处理器。如果仍然失败，资源管理器会提醒集群管理员并使节点脱机，以避免资源管理器再次分配它。图 15-18 是一个具有代表性的序言程序检查例子的伪代码。

```
if [mpss status is stopped]
    service mpss restart
    service ofed-mic start
fi
if [(micctrl -s | grep -c online) < #coprocessors]
    service ofed-mic stop
    micctrl -Rw
    service mpss start
    service ofed-mic start
fi
if [(micctrl -s | grep -c online) < #coprocessors]
    mail -s "Coprocessor boot failure" admin@org.com
    disable node from resource manager and exit
fi
```

图 15-18 TORQUE 序言程序的协处理器“健康”检查

协处理器上允许安装 NFS 共享，有几个这样的共享需要安装。建议各个协处理器共享 Intel Parallel Studio 的编译器套件，这样用户就不必复制各个库。例如在 Beacon 中，/global/opt 文件系统被导出到协处理器上，这个文件系统中包含 Intel 的编译器、多种调试工具、应用程序，以及专门为协处理器创建的一些库。用户库构建和管理的更多相关信息参见 15.8 节。另一种从主机导出的非常有用的文件系统可以是任意高性能文件系统，只要这些文件系统是可用于系统或用户的 NFS 核心区域的。这是非常有必要的，因为如果用户应用程序的输出数据是被写入协处理器的文件系统的，那么输出数据将会丢失。协处理器的

文件系统是易失性的（即基于内存的），一旦作业完成，协处理器重新启动后，存储的数据将被重置。当重导出网络文件系统到协处理器时，导出选项是非常重要的。为了避免增加一层可能的非一致性缓存，作者推荐在每次通过 NFS 重导出网络文件系统时，都要使用 sync 导出选项。

协处理器有几个不同的用户管理方法。默认方式是所有用户都使用“micuser”账户，但是这种方法在多用户环境不能很好地扩展。Intel MPSS 3.2 及以上版本能够利用 micctrl 命令直接动态地添加和删除本地协处理器的用户。图 15-19 说明了使用此选项的语法。

```
micctrl --useradd=<name> --uid=<uid> --gid=<gid> \
[--home=<dir>] [--comment=<string>] [--app=<exec>] \
[--sshkeys=<keydir>] [--nocreate] \
•[--non-unique] [mic card list]
```

图 15-19 micctrl 的 --useradd 命令

在 Intel MPSS 3.2 版之前，管理用户的常用方法是使用类似于图 15-20 中的序言程序脚本，使得可以在作业持续期间访问用户。

```
getent passwd $user | sed "s:./.$user:/User:" | \
sed 's:/bin/.*/sh:/bin/sh:' | ssh ${node}-%mic \
"cat >> /etc/passwd"
getent group $group | ssh ${node}-%mic "cat >> /etc/group"
scp $SSH_KEY_LOCATION/id_rsa* to micX:/$user/.ssh
```

图 15-20 TORQUE 序言程序的内容

在 Beacon 项目中，主机端证书需要通过 LDAP 审查，因此在运行 micctrl 命令或直接访问 /etc/ passwd 文件之前，以上两种方法都需要运行和解析一个 LDAP 查询。如果使用的是 micctrl 方法，则会为用户创建默认 .profile。然而，因为之前已经根据用户当前的环境生成了一个配置文件，所以可以使用这个原始版本。

Beacon 的序言程序步骤结束后，下一步是在协处理器上安装导出的 NFS 文件系统。Intel MPI 还需要向协处理器上复制两个二进制文件，并放置在用户的临时目录中。最后，先前生成的 ssh 密钥也需要复制到协处理器上。图 15-21 显示了这些步骤。

```
scp $tmp/profile micX:/etc/profile
ssh micX "mount nfs shares exported from host"
scp $IMPI/mic/bin/{mpexec.hydra,mpi_proxy} \
micX:$tmp/bin/
scp $SSH_KEY_LOCATION/id_rsa* to micX:/$user/.ssh
```

图 15-21 TORQUE 序言程序的内容

15.5.3 尾声程序

用户的工作结束后，尾声程序将负责清除系统上的用户进程，然后将其系统恢复为干净的状态，以为下一个用户做准备。首先，卸载 NFS 文件系统，以保证 NFS 服务器处于干净的状态。然后重新启动 Intel Xeon Phi 协处理器，以确保整个协处理器处于干净的状态。虽然为尾声程序的运行时间增长了约 60s，但它是一个确保协处理器处于干净状态的方便方法，从而为下一份作业做准备。最后，尾声程序脚本清理用户在主机上创建的目录并退出。

图 15-22 概述了整个过程。

15.5.4 TORQUE/ 协处理器集成

4.2.0 及更高版本的 TORQUE 可以集成到 Intel 协处理器卸载基本库。启用后，pbs_mom 守护进程会自动检测协处理器并定期查询相关参数，如内核数、线程数、最大时钟频率、内存总量、可用内存以及系统平均负载。然后，通过 PBS 调度器 API 可以使这一信息变得可用。当此功能启用后，用户还明确地向部署有协处理器的节点发送请求（例如，“qsub-lnodes=2:mics=4...”）。这个功能在 Beacon 上启用了，但当前没有应用于生产。如果集群中各个节点的协处理器数目不同，或只有一部分节点有协处理器，那么这个功能会非常有用。

```
kill all $user processes
umount NFS_shares
service ofed-mic stop
micctrl -Rw
service ofed-mic start
rm /tmp/$user_stuff
```

图 15-22 TORQUE 尾声程序

15.5.5 Moab

TOEQUE 是用于支持外部调度器的。Adaptive Computing 公司的 Moab 工作负载管理器是最常用的调度器之一。与 TORQUE 的序言程序和尾声程序的工作不同，Moab 不需要知道主机上是否有 Intel Xeon Phi，只要所需的调度策略是基于完整节点进行分配的（即节点之间不共享作业），这也是基于加速器 / 协处理器系统的典型分配策略。

15.5.6 提高网络局部性

通信时需要经过的中间节点数越少，MPI 应用程序的性能也越好；在基于 InfiniBand 的系统中，这通常意味将任务放置于相同的叶交换机或模块下。Moab 可以通过节点集来完成这一目标。通过为节点赋一个可以说明节点所在位置的特征值，然后配置 Moab 使其用这些特征值来代表节点集，用于完成这个任务。在没有最优选择时，可以对一个节点赋多个值，以提供多个位置选择。例如，下面的 Moab 配置为完成作业进行节点作业分配时，首先尝试打包多个节点为一个子集合，然后再将子集合进行打包，形成一个完整集合，最后完成在整个机器的打包（见图 15-23）。

```
# Node sets
# group jobs first within cages, then within racks
# then within the whole system
NODESETPOLICY          ONEOF
NODESETATTRIBUTE        FEATURE
NODESETLIST             r1c1,r1c2,r1c3,r1c4,r2c1,\
                        r2c2,r2c3,r2c4,r4c1,r4c2,\
                        r4c3,r4c4,r1,r2,r4,beacon2
NODESETISOPTIONAL       FALSE
```

图 15-23 Moab 节点设置

15.5.7 Moab/ 协处理器集成

Moab 7.2.0 及之后版本支持通过 TORQUE 来完成低层次的 Intel Xeon Phi 打包。如果 TORQUE 完成了协处理器集成，Moab 就可以根据是否有协处理器来进行调度，并且可以在节点的事件日志中记录协处理器参数。

15.6 健康检查和监控

Intel 提供了在 Intel Xeon Phi 协处理器上部署 Ganglia 所需的 RPM 和相关文档。Ganglia 是一个非常有用的工具，它可以收集统计数据并监控整个集群。作者在 Beacon 上使用 Ganglia 来完成集群的监控和健康检查。在 15.12 节中可以找到 Ganglia 的项目页面。

目前，在协处理器上安装、运行 Ganglia 的方法有两个。其一，如《Intel MPSS 用户指南》中所提到的，在协处理器启动时，通过脚本用 Intel 提供的预生成的 RPM 进行安装。在协处理器所在的主机上执行基于 Python 的 SimpleHTTPServer，然后通过 zypper 向协处理器发布包。使用 Python 在 HTTP 服务器上发布 RPM 时，对于工作站环境，效果会很好，但是对于集群系统，扩展性就不那么好了。因为当有太多请求时，发布文件的 Python 进程可能会频繁停止或挂起。作者建议使用精简的 Web 服务器（如 Apache），用户可以用自己创建的 zypper 进行 RPM 发布。这样，通过配置管理器管理 httpd 进程的过程也会更简单。

在安装 Ganglia RPM 包时，遇到的另一个问题是：/var/mpss/etc/ganglia 中的 gmond.conf 文件会被覆盖。建议将配置文件重命名为其他名字，比如 ganglia.conf，并在守护程序启动时，将该文件指定给 gmond。说明文档还建议使用 ssh 来登录协处理器，然后手动启动 gmond 进程。但是，我们更愿意修改《MPSS 用户指南》文档中所列出的 install-service 脚本，然后将 install-service 安装服务脚本移至 /var/mpss/common/etc/init.d/install-service。同时，需要在 /etc/rc3.d/ 目录下建立相应的软链接，以便协处理器在开机时自动将它载入。图 15-24 给出了相应的例子，其中，\$HOST 应该被替换为该 Webserver 所在主机的位置。

```
zypper ar http://$HOST:8000 klom-mpss-3.2.3
zypper --gpg-auto-import-keys -n --no-gpg-checks \
install ganglia
zypper --gpg-auto-import-keys -n --no-gpg-checks \
install mpss-ganglia
/usr/sbin/gmond -c /etc/ganglia/ganglia.conf
```

图 15-24 Ganglia 安装服务

Intel MPSS 还为我们提供了另一种方法来定制协处理器的文件系统，即 micctrl 的覆盖选项 micctrl--overlay。《MPSS 用户指南》中有该方法的详细介绍。在协处理器上安装 Ganglia 时，使用覆盖选项的安装会更持久，但这个方法的文档不如前面所讨论方法的文档详细。

一旦选择好安装方法，完成所有的安装步骤，并正确配置 Ganglia，Web 前端将显示每个协处理器和主机当前的使用情况。但是，为了使用 Ganglia，主机也需要配置。访问 15.12 节中的 Ganglia 项目页面链接，以了解如何在主机上安装 Ganglia，以及监控系统如何工作。

Ganglia 另一个非常有用的功能是测量协处理器的 CPU 参数。但是，在 ganglia.conf 配置文件下，这个功能是默认关闭的，因为这个功能会使一些应用程序出现轻微的性能损失。出于这个原因，在运行 Ganglia 时，Intel 不建议启用 CPU 指标检测功能。Beacon 内部测试结果显示，启用 CPU 指标检测并未造成任何负面影响，但是在汇集 CPU 指标数据时会产生一些问题。因此，推荐的方案是在提交作业时关闭 Ganglia 的资源和工作负载管理器。

目前，在 Beacon 上进行的工作是将 Ganglia 统计的指标数据存储在统计数据库中，这样系统管理员就可以监控并汇报 Intel Xeon Phi 协处理器的使用情况，并决定如何进行 CPU

和协处理器之间的作业分配。这些使用情况统计信息也可以提供给计算机科学家，以帮助他们监控用户应用程序的效率，并决定是否需要调优以提高性能。这样，计算机科学家就可以主动了解用户应用程序的效率情况，并提供支持。一些用户并不知道他们的应用程序效率低下，浪费了 CPU 时间。我们的目的是帮助用户在更短 CPU 时间内更快地运行出结果，而且这也给了系统更多空闲时间来运行更多的作业。

Keeneland 项目在基于 GPU 的加速器的系统上部署了类似的系统，而其中的灵感正是来自于上述工作。15.12 节给出了 Keeneland 项目的更多相关信息。

15.7 常用命令脚本化

在 Beacon 项目中还创建了另外一些脚本，用于将一些常用命令（如 ssh 和 mpiexec）打包，从而使用户环境更加友好。例如，打包 ssh 指令是很重要的，因为这样可以避免用户在每次调用 ssh 命令时都要添加 `-i $TMPDIR/.micssh/micssh-id_rsa`。需要添加的那部分内容让这个常用的命令显得非常冗长。图 15-25 是打包了 ssh 命令的 micssh 脚本。

```
#!/bin/sh
if [ -n "$PBS_JOBID" -a -n "$TMPDIR" ]
then
    if [ -r $TMPDIR/.micssh/micssh-id_rsa ]
    then
        SSH_ARGS="-i $TMPDIR/.micssh/micssh-id_rsa"
    fi
fi
isroot=$(echo $* | grep -c root)
if [ $isroot -gt 0 ]
then
    echo This program may not be used as root
    exit -1
fi
export SSH_ARGS
ssh $SSH_ARGS $*
```

图 15-25 micssh 脚本

通过简单地改变 micssh 脚本的最后一行，即将 `ssh SSH_ARGS*` 替换为 `scp SSH_ARGS*`，就可以将 micssh 脚本转换为 micscp 脚本。该 micscp 脚本仅适用于小规模跨节点和协处理器的文件移动，当需要将文件分发到大量节点时，该脚本将无法正常工作。例子中的 allmicput 可以让用户轻松地将文件复制到调度器为其作业分配的各个协处理器。

ssh 的另一种打包方式是将类似于图 15-26 中的内容复制到用户的 `~/.ssh/config` 文件中。在 Beacon 中，这部分内容如图 15-26 所示。

```
Host beacon0??-mic?
    IdentityFile ~/.micssh/micssh-id_rsa
```

图 15-26 ~/.ssh/config 文件

micmpiexec 脚本（如图 15-27 所示）可用于执行很多重要的任务，如发送用户的 `$LD_LIBRARY_PATH` 和其他一些环境参数（如 `$IMICROOT`）。`$LD_LIBRARY_PATH` 是存储 Intel Xeon Phi 协处理器编译器库的路径。`$LD_LIBRARY_PATH` 变量是 Intel MPI 根安装目录，`$MKLMICROOT` 是协处理器数学核心库文件的路径。重要的是，把这些路径发送到协处理器后，MPI 作业环境就加载完成了。当用户调用 `module` 命令来改变环境时，这些路

径也可能跟着改变。例如，它们不应该在协处理器的 `/etc/profile` 中静态地设置。相反，应该在协处理器的 `/etc/ssh/sshd_config` 文件中增加图 15-27 中所示内容，这样 `ssh` 的守护进程就能够一直接收这些环境变量。

```
AcceptEnv MIC_LD_LIBRARY_PATH MIC_ENV_PREFIX
```

图 15-27 `/etc/ssh/sshd_config` 设置

`micmpiexec` 脚本中也包含用于传递调试器语法的内容，这对于用正确的命令行参数启动 `mpiexec` 非常必要。图 15-28 中的 `micmpiexec` 脚本能够完成这部分工作。

```
#!/bin/sh
if [ -n "$PBS_JOBID" -a -n "$TMPDIR" ]
then
    if [ -r $TMPDIR/.micssh/micssh-id_rsa ]
    then
        MPI_ARGS="-bootstrap ssh -bootstrap-exec mics\
micssh -genv LD_LIBRARY_PATH $TMPDIR/lib:\
$IMICROOT:$I_MPI_ROOT/mic/lib:$MKLMICROOT:\
$LD_LIBRARY_PATH"
    fi fi
export MPI_ARGS
export EXEC=""
if [ $1 = "-ddt" ]
then
    shift
    EXEC="ddt-client $DEBUGGER_OPTIONS"
elif [ $1 = "-tv" ]
then
    shift
    EXEC="totalview -mmic -args"
fi
if [ -e $TMPDIR/bin/mpiexec.hydra ]
then
    $EXEC $TMPDIR/bin/mpiexec.hydra $MPI_ARGS $*
else
    $EXEC mpiexec.hydra $MPI_ARGS $*
fi
```

图 15-28 `micmpiexec` 脚本

15.8 用户软件环境

Beacon 的异构环境为安装用户所需的软件带来了挑战。幸运的是，在 Beacon 系统的发展过程中，无论是本地的解决方法还是 Intel 工程修复都为此提供了很大的帮助。本节会对发现的问题及其解决方案进行讨论，还将详细介绍 Beacon 当前的用户软件环境。

Beacon 软件栈的结构是通过 `swtools` 生成的，`swtools` 包括由橡树岭核心计算设备维护的软件安装管理包和前面提到的模块系统。这给维护软件的人提供了一个自动化系统，以用于自动重建应用程序或库，自动生成基于 Web 的文档和软件列表，并测试库或应用程序，以确保它们能在所有平台上正常工作。它能够用于确保软件默认选择的是最稳定的版本，其他版本都将被列为“试用的”或“过时的”。最后，它还包括软件说明文档，用于说明软件用途以及支持该软件的平台。当然，它还包括一个自动检查更新的进程。

在异构系统中使用上述结构创建软件栈时，首先必须解决的一个难题是，软件必须分别在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上进行编译。当 Beacon 首次投入生产时，所需软件树分为 MIC 部分和 XEON 部分，以使这种区别更加明显。这样区分开来虽然简化了

用户为初始模式和卸载模式的应用选择正确模块的过程，但它也使在对称模式下的编译和运行变得复杂。在对称模式下，软件分别为协处理器和处理器编译好，然后在通用的 MPI 通信器下，把每个软件分配到相应的硬件设备上运行，但是这些模块不能同时加载。

经过测试，NICS 科学计算小组确定了编译器在为一种模式或另一种模式进行编译时，能够跳过格式不正确的库。在 Beacon 上进行的实验确定了多种模式文件可以组合成一个混合模式树。图 15-29 中的命令将两组库或者二进制文件的路径分别加载到 LD_LIBRARY_PATH 和 PATH 环境变量中。

```
module load example-lib
```

图 15-29 模块命令示例

另外，MIC_LD_LIBRARY_PATH 环境变量中包含了模块库协处理器版本的路径。它的存在为区分 Xeon 和 Xeon Phi 库带来了很大的便利。同时，在需要所有库且库的顺序非常重要时，它可以用于将一条路径附加到其他路径上。对于用户只加载一个模块文件并且无缝地在处理器、协处理器或同时在两者上运行，这起着非常重要的作用。

在混合库的环境下，编译时会出现“正在跳过错误类型的库”的警告，但是正确的库在编译和运行时可以正确使用。如果用户想要在对称模式下运行代码，在编译和运行时，多个编译器通常可以访问两组库。

目前，没有自带的 Intel 编译器，但是 MPSS 提供了 GNU 工具链以在 Intel Xeon Phi 上直接使用。对于协处理器，这个 GNU 工具链只用来创建工具，而不能用于应用程序，因为它没有针对协处理器的指令集进行优化。在协处理器上运行的软件是使用 Intel 工具链构建的，为了适应这一情况，在主机上，代码将采用有 -mmic 选项的交叉编译。复杂代码的交叉编译需要所有交叉编译选项都正确设置才可以成功，这样才能避免以协处理器为目标的代码的编译不会试图作为中间步骤在主机上运行，从而导致安装或配置失败。如果软件必须编译和运行一段程序去获取一些值以完成安装，交叉编译也可能会中途停止。上述情况对于处理器和协处理器都需要繁琐的编译过程：在主机上运行处理器可执行文件（或在协处理器上运行合适的目标版本，这需要发送和接收代码与数据），将生成的文件传回协处理器编译，编辑输出文件以确保它用于协处理器执行或者进一步的编译，然后仅在主机上恢复协处理器编译。

Beacon 上的软件栈包含了 30 个预装软件供所有用户使用，其中包括库和应用程序，如 OpenFOAM 和 R，以及常用的开发库，比如 Boost、GlobalArrays、MAGMA、HDF5、NETCDF 和 Metis 等。此外，除了 Intel Parallel Studio 企业版中所提供的，性能分析工具还包括了 FPMPI 和 MPIinside 等。系统管理员将每个模块设置为一个默认版本，当然，用户可以选择他们想要的任何版本。

除了 Intel、GNU 和 CAPS 编译器（可以将 OpenACC 代码转化为 OpenCL 代码，OpenCL 代码可以编译为在协处理器上运行），Beacon 还使用了 Rogue Wave 的 TotalView 软件调试器、Allinea 的 DDT 调试器以及 MAP 分析器。为了使用 Intel 的 IMPIS.0、OpenMPI 以及 MVAPICH，近期增加了提交一个请求不同 MPI 实现的作业的功能。

15.9 今后的方向

展望未来，在集群上部署 Intel Xeon Phi 协处理器最主要的问题是软件部署和管理的可能

扩展性。早期 Intel MPSS 版本的安装和配置主要集中在工作站环境，但在几十或几百个节点上这是难以安装和自动化的。幸运的是，通过社区反馈、直接参与合作 Beacon 项目和其他 HPC 部署，Intel 已经提出了一些 MPSS 改进措施，使 MPSS 的模型变得更加友好，例如在通过配置管理系统的自动化安装，以及无状态操作系统部署（如 NFS 根）等方面。Intel MPSS 还计划完成更多的工作以支持集群的 HPC 环境，以及提供配置文件的透明性，这将有助于减少管理员在之后部署工作中的开销。另外，随着 Intel MPSS 逐渐成熟，API 和配置文件的格式变更也将越来越少，这将降低更新版本所带来的困难和风险。

作者预计，代号为“Knights Landing”（KNL）的下一代 Intel Xeon Phi 硬件发布后，与目前可用的 Intel Xeon Phi 协处理器相关的软件管理问题，都将会得到大大改善。插槽式 KNL 将能够引导一个现成 Linux 操作系统版本，如 RHEL。由于在同一节点内管理主机和协处理器，因此将 Intel Xeon Phi 处理器作为一个标准的可启动平台，可能会缓解目前的许多难题，如资源管理系统、监测、一般管理和自动化等。

15.10 总结

相对于一般的 CPU，Intel MIC 架构（以目前的 Intel Xeon Phi 协处理器为代表）对于高度并行的应用具有明显的性能优势，并且相对于其他加速器和协处理器设备，在可用性方面有较大的提升；但是这也使得系统管理更加复杂，尤其是对于集群环境。多年来，通过与 Intel 一起部署和操作多个配备有开发和生产版本协处理器的超级计算机平台，NICS 积累了大量经验，并提出了许多方法和最佳实践，用于解决大部分与架构相关的操作难题，以使设备部署和操作过程更灵活，并为用户提供更友好的操作环境。这些方法和实践将为更广泛的计算社区降低门槛，提高用户的工作效率。随着新的 Intel MPSS 栈和新一代 Intel MIC 架构的出现，这些集群管理技术还将不断完善和改进，以确保与一般的超级计算社区持续关联。

15.11 致谢

作者要感谢来自 NICS 的 George Butler 和 Jesse Hanley。他们的见解、经验和努力为这项工作做出了很大的贡献。

本章内容基于由美国国家科学基金会（编号 1137097）和美国田纳西大学资助的 Beacon 项目的工作成果。本章中的任何意见、研究成果和结论或建议都代表作者本人，并不一定反映美国国家科学基金会和美国田纳西大学的意见。

15.12 更多信息

- NICS 宣布与 Intel 的战略合作，<http://www.nics.tennessee.edu/intelpartnership>。
- 视频：Intel 的 Knights Corner 能够在一个芯片上实现 1TFLOPS，SC11，<http://insidehpc.com/2011/11/16/video-intels-knights-corner-does-1-teraflop-on-a-single-chip-at-sc11/>。
- Intel MIC 架构上的科学应用，<https://www.youtube.com/watch?v=TOVokMClr5g>。
- nfsroot——Linux 集群上的 NFS 根支持，<https://code.google.com/p/nfsroot/>。
- Green500 列表——2012 年 11 月，<http://www.green500.org/news/green500-list-november-2012>。
- 合作创造最节能的超级计算机，<http://blogs.intel.com/intellabs/2012/11/14/green500-sc12/>。

- Intel 众核平台软件栈 (MPSS), <http://software.intel.com/en-us/articles/intel-many-core-platform-software-stack-mpss/>。
- TORQUE 资源管理器——Adaptive Computing 公司, <http://www.adaptivecomputing.com/products/open-source/torque/>。
- Moab HPC 套件的基本版本——Adaptive Computing 公司, <http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/>。
- Modules——软件环境管理, <http://modules.sourceforge.net/>。
- Lmod: 环境模块系统, <https://www.tacc.utexas.edu/tacc-projects/lmod/>。
- pdsh——并行分布式 Shell, <https://code.google.com/p/pdsh/>。
- MPSS 用户指南, http://registrationcenter.intel.com/irc_nas/4245/MPSS_Users_Guide.pdf。
- Ganglia 监控系统, <http://ganglia.sourceforge.net/>。
- Vetter, J.S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., Yalamanchili, S., 2011. Keeneland: bringing heterogeneous GPU computing to the computational science community. IEEE Comput. Sci. Eng. 13, 90-95.
- SWTools, <https://www.olcf.ornl.gov/center-projects/swtools/>。
- NICS 的 Beacon 资源页面, <https://www.nics.tennessee.edu/beacon>。
- TACC Stampede 用户指南, <https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide>。
- 本章提到的脚本和配置文件, <http://lotsofcores.com/>。

在 Intel Xeon Phi 协处理器上支持集群文件系统

Michael Hebenstreit

美国, Intel 公司

如果用户不仅仅以卸载模式使用 Intel Xeon Phi 协处理器,而是将其作为集群上的一个计算节点进行访问,就需要进行一些特定配置:1) 解决网络布局;2) Intel Xeon Phi 协处理器能够使用集群上安装的文件系统。当完成恰当的配置后,用户就可以从集群上的任意节点直接访问协处理器,并找到同标准节点类似或相同的应用环境(如集群文件系统)。本章将讨论如何配置协处理器,以便将计算节点上的协处理器在整个集群范围内被看作标准计算节点。特别是,这将使应用程序可以在所有可能的用户模式下运行 MPI 代码。这些用户模式包括:1) 主机处理器通过卸载模式使用协处理器;2) 作为原生协处理器独立运行 MPI 代码;3) 主机处理器和协处理器同时运行 MPI 代码。

为说明我们的目标,本章使用 Intel MPI 库的一个示例在配备主机节点和协处理器“节点”的集群上运行 MPI 代码。MPI 集群的运行模式允许直接增加一个协处理器运行 MPI 程序。步骤如下:

- 设置 Intel MPI 库和 Intel C/C++ 编译器环境:

```
$ . /opt/intel/impi/latest/bin64/mpivars.sh
$ . /opt/intel/compiler/latest/bin/iccvars.sh intel64
```

- 为主机处理器编译代码:

```
$ mpiicc -o mpitest test.c
```

- 为协处理器编译相同的代码:

```
$ mpiicc -mmic -o mpitest.mic test.c
```

- 创建包含主机和协处理节点名称的文件:

```
$ echo test1-mic0 > hostfile
$ echo test2-mic0 >> hostfile
$ echo test1 >> hostfile
$ echo test2 >> hostfile
```

- 设置协处理器选项并运行代码:

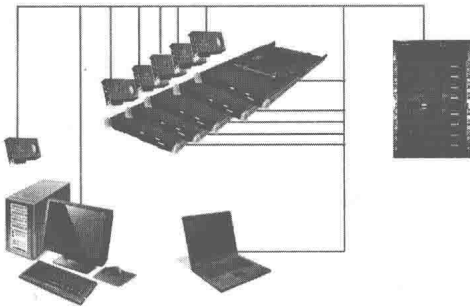
```
$ export I_MPI_MIC=enable
$ export I_MPI_MIC_POSTFIX=.mic
$ mpirun -bootstrap ssh -perhost 1 -f hostfile -n 4 ~/mpitest
```

本站将描述必要的协处理和集群配置步骤,以便使这个过程尽量简单。

16.1 网络配置概念和目标

相对于协处理器被工作站内的单个用户使用,为协处理器配置网络需要考虑更多因素。特别是,必须要考虑对多用户的影响以及用户对网络连接的期望。将 Intel Xeon Phi 协处理

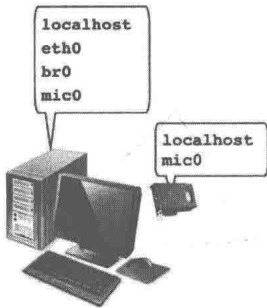
器作为卸载处理器，或者作为独立的原生 SMP 系统，或者作为包含计算节点的集群，相应的灵活性都需要根据系统用户的实际需求进行配置。



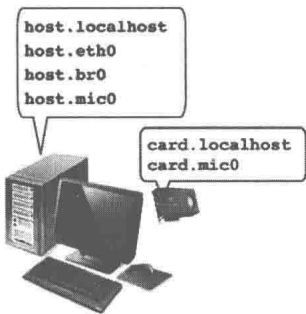
例如，用户期望从自己的工作站或者笔记本电脑通过网络直接连接所有其他系统。用户更期望在无须输入密码的情况下，通过 ssh 连接同一集群上的所有计算节点。这是运行基于 MPI 的程序和更简单脚本的必要前提。

16.1.1 网络选项概览

从网络接口层开始，注意，在 Linux 服务器上安装协处理器时，系统会为每个协处理器创建两个额外的网络接口，如下图所示。



这些接口将会被 Intel 众核平台软件栈（Intel MPSS）使用，以允许协处理器进行 IP 通信。但是，无论是主机还是协处理器接口都称为 mic0。本章使用 host.mic0 和 card.mic0 来区分主机和协处理器定义的接口。



这里有三个基本的网络配置选项。

- 1. 在主机和协处理器间创建一个私有网络。例如，下面的配置将允许主机和协处理器进

行通信，但网络上的其他节点将不能访问该协处理器。

```
host.mic0 192.168.0.1 netmask 255.255.255.0
card.mic0 192.168.0.2 netmask 255.255.255.0
```

2. 配置所有卡监听唯一地址。在为系统的所有协处理器增加显式路由或者安装一个路由守护线程来发现和广播所需要信息的前提下，这个配置就会允许远程系统访问协处理器。作者这里强烈建议使用后者，因为路由表的规模和复杂度将会随系统规模的平方而增长。在一个大型系统上，这会影响网络性能。

3. 在主机系统上部署一个网络桥，为协处理器提供在网络上的可访问性。在这个配置中，无论是 host.mic0 还是 host.eth0 接口都是可以通过 host.br0 设备连接的。除此之外，card.mic0 现在可以自动监听发送到协处理器地址的信息的网络流量。

为了避免发生问题，host.mic0 和 card.mic0 都需要固定的唯一 MAC 地址（机器重启后不变）。例如，对于一个装备两块协处理器的服务器，在网络桥的配置下，交换机将会看到 5 个 MAC 地址。除非每次重启后协处理器的 MAC 地址会发生变化，否则这将不是一个问题。如果这种情况发生，特别是在集群上，交换机或者文件服务器上的缓冲区将会超负荷运转。不过通常这不是一个问题，因为协处理器可以利用它们的序列号来确定一个唯一和正确的 MAC 地址。

16.1.2 设置集群启用协处理器的步骤

1. 确保集群上所有系统都知道各种形式的主机名。这包括主机名、以太网的 IP 地址、InfiniBand 接口上的 IP，对于协处理器，同主机类似。

2. 支持 InfiniBand，需要安装一个合适的 OFED 版本。在本书写作时，这个版本或者是 OFED 1.5.4.1，或者是 OFA-3.5.2-mic。请使用 16.4 节中的下载链接下载最新的 Intel MPSS 版本、兼容的 OFED 版本以及更新的安装信息。

3. 确保已经安装了 Intel MPSS 和协处理器卡固件的最新版本。

4. 根据 Intel MPSS 包中的指导信息为当前 Linux 内核版本重新编译和安装 MPSS 内核模块 Intel 提供了针对 Redhat 和 SuSE* 企业级发行版系统的安装包，但对其更新版本，没有对内核模块进行编译和测试。注意，在 Linux 内核更新之后，MPSS 内核模块也很有可能需要更新。

5. 重新编译和安装 MIC-OFED 内核模块。注意，当 Linux 内核发生更新时，这些模块需要重新编译、安装。需要知道，在重新编译 MIC-OFED 内核模块时，必须要安装 Intel MPSS 的内核模块的 DEVELOPMENT 文件。

6. 配置主机系统使用桥网络设置（见上一节）。下面是为 Redhat* 企业级 Linux 版本 6 系统配置桥 br0 的例子：

```
# cat /etc/sysconfig/network-scripts/ifcfg-br0
DEVICE=br0
TYPE=Bridge
ONBOOT=yes
DELAY=0
NM_CONTROLLED="no"
MTU=9000
BOOTPROTO=dhcp
NOZEROCONF=yes
```


修改 `/etc/sysconfig/network-scripts/ifcfg-eth0`，以使用 `br0` 桥：

```
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
ONBOOT=yes
BRIDGE=br0
MTU=9000
```

确保把原 `eth0` 的地址赋给 `br0`。

7. 利用下面命令为协处理器创建一个基本配置：

```
micctrl --initdefaults
```

8. 或者通过 `micctrl` 或者通过主机操作系统为协处理器配置网络和桥网络。下面是为 Redhat* 企业版 Linux 版本 6 系统进行配置的例子：

```
# cat /etc/sysconfig/network-scripts/ifcfg-mic0
DEVICE=eth0
ONBOOT=yes
BRIDGE=br0
MTU=9000
```

9. 协处理器的网络配置支持静态 IP 地址和 DHCP。配置遵循标准的 Linux 原理。在 Intel MPSS 3.x 上，每个协处理器的网络配置文件位于：

```
/var/mpss/mic[012...]/etc/networking/interfaces
```

同时，对于旧的 Intel MPSS 2.x，这个配置文件使用不同的语法，并位于：`/opt/intel/mic/filesystem/mic[012...]/etc/sysconfig/network`。

10. 使用下面的 `micctrl` 命令序列开启协处理器（`-r` 重启协处理器；`-w` 等待操作的完成，`-b` 启动协处理器）。

```
# micctrl -r
# micctrl -w
# micctrl -b
```

11. 通过 `root` 用户测试能否访问协处理器。如果访问失败，你的 `/root/.ssh` 文件夹可能没有正确安装。

```
# ssh `hostname`-mic0 pwd
/root
```

16.2 协处理器文件系统支持

如果用户不仅仅利用卸载模式使用 Intel Xeon Phi 协处理器，而是将其作为集群上的一个计算节点进行访问，就需要进行一些额外配置。既然我们已经解决了网络布局问题，现在就需要让协处理器需要能够使用集群上安装的文件系统了。

接下来的讨论将详细描述如何让协处理器支持 NFS、Lustre、BeeGFS 和 Panasas PanFS 文件系统。

一旦这些改变完成，用户就可以直接在协处理器上运行 MPI 应用程序了。

16.2.1 支持 NFS

Intel MPSS 为协处理器提供了 NFS v3。使用 NFS 的挑战是正确配置网络。一旦协处理

器可以成功“ping”通，并直接连接上 NFS 服务器，mount 命令的使用和标准 Linux 系统没有差别。

```
test-mic0# mkdir -p /mnt
test-mic0# mount -o rw,nolock 10.101.235.1:/scratch1 /mnt
test-mic0# mount | grep /mnt
10.101.235.1:/scratch1 on /mnt type nfs
(rw,relatime,vers=3,rsz=1048576,wsz=1048576,namlen=25
5,hard,proto=tcp,port=65535,timeo=70,retrans=3,sec=sys,local_lock=none,addr=36.101.235.1)
```

该内核不支持 NFS v4。对 NFS v4 感兴趣的管理人员可以激活 NFS4 并重新编译协处理器的 Linux 操作系统内核。

16.2.2 支持 Lustre 文件系统

协处理器能够支持 Lustre 并行文件系统。为协处理器编译 Lustre 客户端模块和在主机端构建客户端模块相似。Intel MPSS 提供了一个能够构建内核模块的交叉编译器。

对于 Intel MPSS 2.1.x，协处理器的操作系统内核位于 /opt/intel/mic/src/card/kernel，构建指令可使用如下命令：

```
$ export PATH=/usr/linux-klom-4.7/bin:$PATH
$ test -e ./autogen.sh && sh ./autogen.sh
$ ./configure --with-linux=/opt/intel/mic/src/card/kernel \
--disable-server --without-o2ib \
--host=x86_64-klom-linux --build=x86_64-pc-linux
$ make
$ make rpms
```

注意，Intel MPSS v3.1.2 及以上版本已经连接可用的 RDMA 模式。这就意味着这些版本的 Lustre 可以在直接 InfiniBand 支持的前提下进行编译。对于架构的变化必须要注意。对于 MPSS 3.x 版本，微架构名称从 x86_64-klom-linux 变为 klom-mpss-linux。

如果没有 autogen.sh 文件，这将成为一个问题。一个解决方案是修改所有的 config.sub 文件：

```
$ find . -name config.sub -exec \
sed -e 's,x86_64-\* |,x86_64-\* | klom-\*|,' -i {} \;
```

对于 Intel MPSS 3.x，有如下编译命令：

```
$ ./opt/mpss/3.x/environment-setup-klom-mpss-linux
$ test -e ./autogen.sh && sh ./autogen.sh
$ ./configure --with-linux=/SOMEPATH/linux-2.6.38+mpss3.x \
--with-o2ib=/usr/src/ofed-driver \
--host=klom-mpss-linux --build=x86_64-pc-linux
$ make
$ make rpms
```

对于 MPSS 3.2.3，路径 --with-o2ib=/usr/src/ofed-driver 实际涉及主机端的 mpss-ofed-dev-*.x86_64.rpm 包。可能的原因是主机操作系统和协处理器操作系统使用相同的 OFED 版本和 API。

这个模块必须加载到在 Intel Xeon Phi 协处理器上运行的内核中。为简单起见，挂载 (mount) NFS 文件系统为主机和协处理器提供相同路径。一旦 RPM 创建，rpm 目录将发生改变，可使用 rpm2cpio 命令来提取文件。

在主机上：


```
$ cd /NFSPATH
$ rpm2cpio ~/rpmbuild/RPMS/x86_64/lustre-client-mic-2.5.0-
2.6.38.8_gefd324e.x86_64.rpm | cpio -idm
$ rpm2cpio ~/rpmbuild/RPMS/x86_64/lustre-client-mic-modules-
2.5.0-2.6.38.8_gefd324e.x86_64.rpm | cpio -idm
```

使用 root 用户，当没有 InfiniBand 支持时，在协处理器上运行：

```
# INSTALLPATH=/NFSPATH/opt/lustre/2.5.0/ARCHITECTURE
# LIBPATH=/lib/modules/`uname -r`
# for I in $INSTALLPATH/$LIBPATH/updates/kernel/*/lustre/*.ko \
do cp $I $LIBPATH \
done
# depmod -a
# modprobe lustre
# lsmod
# mkdir /mnt
# $INSTALLPATH/sbin/mount.lustre 12.12.11.1@tcp:/lfs08 /mnt
```

使用 MPSS 3.x 并支持 InfiniBand 的系统必须配置 ipoib 接口。既可以使用 Intel MPSS 文档中提供的方法，也可以直接加载驱动程序并配置接口：

```
# /sbin/modprobe ib_ipoib
# /sbin/ifconfig ib0 12.12.12.100 netmask 255.255.0.0
# /sbin/modprobe ibp_sa_client
# /sbin/modprobe ibp_cm_client
```

挂载 Lustre 文件系统，并使用 root 用户在协处理器上执行如下命令：

```
# INSTALLPATH=/NFSPATH/opt/lustre/2.5.0/klom-mpss-linux/
# LIBPATH=/lib/modules/`uname -r`/updates/kernel
# mkdir -p $LIBPATH/net/lustre
# mkdir -p $LIBPATH/fs/lustre
# FILES=`cd $INSTALLPATH/$LIBPATH; ls */lustre/*.ko`
# for I in $FILES
do
ln -s $INSTALLPATH/$LIBPATH/$I $LIBPATH/$I
done
# echo 'options lnet networks="o2ib0(ib0)'" \
> /etc/modprobe.d/lustre.conf
# /sbin/depmod -a
# /sbin/modprobe lustre
# /sbin/lsmod
# mkdir /mnt
# $INSTALLPATH/sbin/mount.lustre 12.12.12.1@o2ib:/lfs08 /mnt
```

注意，在本书写作时，rdma_cm 支持仍然在实验中。在将来的 Intel MPSS 版本中，一些步骤可能就不需要了。

16.2.3 支持 Fraunhofer BeeGFS 文件系统

BeeGFS (原名 FHGFS) 文件系统由 Fraunhofer ITWM 研究所开发，并提供了对原生 Intel Xeon Phi 协处理的支持以及 InfiniBand 的支持。

支持 InfiniBand 需要 rdma_cm 内核模块，这就需要 Intel MPSS v3.1.2 及以上版本。一旦 OFED 和 Intel MPSS 安装运行，BeeGFS 内核模块就可以在协处理器上进行编译。

BeeGFS 源代码可从 <http://www.fhgfs.com/cms/download> 下载，默认安装路径是 `/opt/fhgfs/src/client`，可以非常容易地修改该路径。如变换到 `/opt/fhgs/src/client/fhgfs_client_module/build` 目录（或者用户指定的编译目录）。

下面的 make 命令假设使用默认路径：


```
$ make KDIR=/PATHTO/linux-2.6.38+mpss.../
STRIP=/usr/linux-klom-4.7/bin/x86_64-klom-linux-strip
CC=/usr/linux-klom-4.7/bin/x86_64-klom-linux-gcc
LD=/usr/linux-klom-4.7/bin/x86_64-klom-linux-ld
LDFLAGS="-m elf_klom"
FHGFS_OPENTK_IBVERBS=1
OFED_INCLUDE_PATH=/PATHTO/mpss-ofed/src/kernel/include/
```

这创建了模块：

```
.../src/client/fhgfs_client_opentk_module/build/fhgfs-client-
opentk.ko
```

以及模块：

```
.../src/client/fhgfs_client_module/build/fhgfs.ko.
```

将这些内核模块发送到每个协处理器的文件系统，并使用 Linux 命令加载它们。例如：

```
$ insmod fhgfs-client-opentk.ko
$ insmod fhgfs.ko
```

现在，BeeGFS 可以按照通用方法挂载到协处理器上了。

16.2.4 支持 Panasas PanFS 文件系统

对于 Panasas PanFS 5.0.1.e, Panasas 用户可从官网上为 Intel Xeon Phi 协处理器下载驱动程序和原生工具：<http://www.panasas.com>。

因为 Panasas 允许下载源代码，所以用户可能希望在 Intel Xeon Phi 协处理器环境下直接编译它。这是可能的，但有些复杂。因为这需要创建一个包含 Perl 工具的编译环境，并且去除使用 Intel MMX 汇编指令的少量源代码。

16.2.5 集群文件系统的选择

在很多情况下，Intel Xeon Phi 协处理器会集成到已有的集群中，这时集群文件系统（Cluster File System，CFS）的选择已经决定了。如果集群使用的 CFS 前面没有涉及，用户可以或者使用中间服务器挂载和再导出，或者移植客户软件。根据作者的经验，这项工作听起来非常困难，但它确实如此。

上面提到的 4 种文件系统之间，用户可以在一个理想的集群中同时使用 4 个文件系统。这是因为它们各有优势：

NFS 轻松得到了广泛的支持（甚至在 Windows 上）；使用以太网；单线程性能较低；可扩展性低；用于管理文件系统；用于小集群（<500 个节点）上的数据文件系统。

Lustre 使用以太网和 InfiniBand；单线程性能适中；可扩展性高；用于快速暂存文件系统。

BeeGFS 使用以太网和 InfiniBand；单线程性能高；可扩展性高；用于快速暂存文件系统。

PanFS 使用以太网；单线程性能低；可扩展性高；可靠度高，因此提供高度的数据安全性；用于数据文件系统。

没有方法来确定边界，而且可以改进。例如：PanFS 的单线程性能可以通过使用在 10GE 和 InfiniBand 间的路由器盒进行提升；或者使用恰当的硬件和软件，动态增加 Lustre 的可靠性。

16.3 总结

关于如何构建在网络拓扑中将 Intel Xeon Phi 协处理看作一个独立的标准计算节点的集

群，本章提供了必要的信息。网络配置和集群文件系统都需要进行修改。有了这些组件和修改，在协处理器上运行分布式 MPI 程序就会非常简单，只须按照本章提供的步骤重新编译即可。至于将程序优化以达到特定的计算性能，并考虑在处理器和协处理器上优化的不同，这是另外一个话题。

16.4 更多信息

Intel MPSS 软件下载：

- <https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>

NFS：

- http://en.wikipedia.org/wiki/Network_File_System

Lustre：

- http://wiki.lustre.org/index.php/Main_Page
- <http://www.intel.com/content/www/us/en/software/intel-solutions-for-lustre-software.html>

Panasas：

- <http://www.panasas.com/>

BeeGFS：

- <http://www.fhgfs.com/cms/>

Clustering 介绍：

- <http://www.hpcgeeks.com/index.php/hpc-hardware/hpc-clusters/5-build-your-own-cluster>
- http://www.admin-magazine.com/HPC/Articles/real_world_hpc_setting_up_an_hpc_cluster
- <http://www.wikihow.com/Build-a-Supercomputer>

MPI：

- <http://www.mpi-forum.org/docs/>

Linux 网络桥：

- <http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge>

NWChem: 大规模量子化学仿真

Edoardo Aprà^{*}, Karol Kowalski^{*}, Jeff R. Hammond[†], Michael Klemm[‡]

^{*} 美国, 太平洋西北国家实验室; [†] 美国, Intel 公司; [‡] 德国, Intel 公司

17.1 引言

自 20 世纪 30 年代以来, 为了能够精准地研究分子的电子结构, 研究人员提出了基于量子方程的研究方法。随着近 20 年来新算法的快速发展, 将高阶多体方法应用在具有挑战性的分子过程精确模拟问题上成为可能。大量证据表明, 电子间的相互作用对于准确地表述化学反应的性质、分子特性和光与物质的相互作用是不可或缺的。可靠的中型分子系统基准测试方法同样为多尺度方法在空间尺度进行高阶准确传播提供了一种尝试。其中一些方法可以有效利用计算资源, 因为它们具有高计算复杂度和适当的数据粒度。如广泛的耦合簇 (coupled cluster, CC) 就属于这类方法。最近的一些 CC 实现清晰地展示了 CC 形式在众核计算机架构上的可扩展性。在本章中, NWChem 提供了一系列张量收缩引擎 (Tensor Contraction Engine, TCE), 这些算法由不同 CC 方法实现这些 CC 方法又充分利用众核计算机架构。

基于 GPU 或者 Intel 众核架构的混合架构提供了另一种降低高阶 CC 算法求解时间的方法选择。Intel Xeon Phi 协处理器上对 CCSD (T) 方法的实现就是一个很好的例子。因此 CCSD 在精确计算化学领域中被视为一条“黄金标准”。

对于领域专家来说, 是除获得高性能之外, 将这些模拟工具在硬件上高效实现是实现成功的关键需求。在推出时, Intel Xeon Phi 协处理器承诺是比传统的协处理器更加有效的计算部件, 同时也更易于向协处理器移植代码。根据我们在 NWChem 化学包方面的经验, 基于 Xeon Phi 的承诺, 我们充分利用由 MIC 平台软件栈、Fortran 的 Intel Composer XE 和 Intel VTune Amplifier XE 所组成的工具链优化代码。我们的目标是不重写现有 Fortran 代码, 以防止这些代码不适用于 Intel Xeon 平台。同时, 我们希望继续开发和优化 Fortran 代码, 使得 Intel Xeon 处理器和 Intel Xeon Phi 协处理器都获得更好的性能 (至少不会降低它们的性能)。

在本章中, 我们使用高性能卸载解决方案获得高产出率的代码。这种高性能卸载方法在最近 Top500 排名前 20 的系统中显示出非常的高并行性能和可扩展性。在我们的研究中, 定义了一种新的方法, 开发者可以使用这种方法决定代码中哪些部分适合卸载到协处理器上进行计算, 以及需要哪些步骤将这些代码迁移到 Intel 协处理器上。我们的方法为卸载位置和计算机内核优化提供了必要的输入信息。下面先快速介绍理论背景和 NWChem 的基本架构。

17.2 回顾单线程 CC 形式

我们先简要描述一下单线程 CC 形式。我们也将着重讨论方程的数值形式和并行策略。CC 理论建立在存在一个 Slater 行列式 $|\Phi\rangle$ 的假设上, 这个行列式可视为参考函数, 其能够通过波函数 $|\Psi\rangle$ 提供一个基态电子态零阶的合理描述。通常情况下, 波函数被选作 Hartree-

Fock (HF) 行列式。CC 参数化中的波函数是以指数拟合的形式出现的。

$$|\Psi\rangle = e^T |\Phi\rangle \quad (17-1)$$

这里簇操作符 T 只由链接图表示。引入簇操作符 CC 方程的典型方式是将式 (17-1) 写成薛定谔方程的形式:

$$He^T |\Phi\rangle = Ee^T |\Phi\rangle \quad (17-2)$$

将式 (17-2) 两边都左乘 e^{-T} , 可得

$$e^{-T} He^T |\Phi\rangle = E |\Phi\rangle \quad (17-3)$$

使用以下 Baker-Campbell-Hausdorff 引理

$$e^{-B} A e^B = A + [A, B] + \frac{1}{2!} [[A, B], B] + \frac{1}{3!} [[[A, B], B], B] + \dots \quad (17-4)$$

能够将式 (17-2) 写成如下形式:

$$(He^T)_C |\Phi\rangle = E |\Phi\rangle \quad (17-5)$$

这里下标 “C” 表示给定运算符的链接图。通过相对于参考行列式 $|\Phi\rangle$ 将式 (17-5) 投影到各个可能激发的立体基阵 $|\Phi_{i_1 \dots i_n}^{a_1 \dots a_n}\rangle$, 我们得到簇振幅的去耦合等式

$$\langle \Phi_{i_1 \dots i_n}^{a_1 \dots a_n} | (He^T)_C |\Phi\rangle = 0 \quad (17-6)$$

从能量角度, 通过投影式 (17-5) 到参考函数得到如下形式:

$$E = \langle \Phi | (He^T)_C |\Phi\rangle \quad (17-7)$$

上述等式用于确定簇振幅和对应的能量。我们应该能够注意到, 对于 T 操作符, 第一步非线性等式需要迭代解决, 接着应该使用已知的簇振幅计算能量。

在实际的应用中, 簇操作符可由多体展开式近似, 而多体展开式在特定激发能级 m_A ($m_A \ll N$, 其中 N 被指定为系统中互相关的电子数量) 被截断:

$$T = \sum_{m_A}^{n=1} T_n \quad (17-8)$$

其中 T_n 是簇操作符 T 的一部分, 当 T 应用于参考函数时, 便会产生 N 元组激发。最常见的近似是 CCSD 方法 (单、双 CC 方法), 其中簇操作符是通过单激发 (T_1) 和双激发 (T_2) 多体元素所定义的。这推导出了如下 CCSD 波函数:

$$|\Psi_{\text{CCSD}}\rangle = e^{T_1 + T_2} |\Phi\rangle \quad (17-9)$$

其中 T_1 和 T_2 代表单激发 (t_a^i) 和双激发的 (t_{ab}^{ij}) 簇振幅以及对应的创建/毁灭操作符 (X_p^+/X_p):

$$T_1 = \sum_{i,a} t_a^i X_a^+ X_i \quad (17-10)$$

$$T_2 = \frac{1}{4} \sum_{i,j,a,b} t_{ab}^{ij} X_a^+ X_b^+ X_j X_i \quad (17-11)$$

$i, j \dots (a, b \dots)$ 下标表示被占据的 (未被占据的) 旋转轨道在参考函数 $|\Phi\rangle$ 中的下标。簇振幅的标准 CCSD 等式可以从薛定谔方程的链接形式中得到, 薛定谔方程是通过投影到单激发 ($|\Phi_i^a\rangle, |\Phi_i^a\rangle = X_a^+ X_i |\Phi\rangle$) 和双激发 ($|\Phi_{ij}^{ab}\rangle, |\Phi_{ij}^{ab}\rangle = X_a^+ X_b^+ X_j X_i |\Phi\rangle$) 的立体基阵而定义的:

$$\langle \Phi_i^a | (He^{T_1 + T_2})_C |\Phi\rangle = 0 \quad \forall i, a \quad (17-12)$$

$$\langle \Phi_{ij}^{ab} | (He^{T_1 + T_2})_C |\Phi\rangle = 0 \quad \forall i, j, a, b \quad (17-13)$$

其中, H 是电子哈密顿函数操作符。一旦从式 (17-12) 和式 (17-13) 中确定簇振幅, CCSD 能量就可以从下面的表达式计算出来:

$$E = \langle \Phi | (He^{T_1 + T_2})_C |\Phi\rangle \quad (17-14)$$

使用 diagrammatic 技术, 可以很容易确定式 (17-12) 和式 (17-13) 的代数结构以及对应的等式计算复杂度, 其中计算复杂度正比于 $n_o^2 n_u^4$ (n_o 和 n_u 分别表示占据和未被占据的旋转轨道)。但是, 多数情况下准确获得 CCSD 形式并不足以提供所谓的化学准确性, 化学准确性被定义为误差低于 1kcal/mol。为了得到化学准确性, 必须包含三激发 (用 T_3 操作符表示)。

直接在簇操作中包含 3 体效果 (T_3) 会导致高计算规模 (约等于 $n_o^3 n_u^5$) 和高内存需求 (约等于 $n_o^3 n_u^3$), 这些都阻碍了 CCSDT (单、双、三 CC 方法) 的计算, 即便是对小分子系统的计算也如此。为了降低 CCSDT 的规模而不降低精度, 过去几年已经有了很多方法, 其中以微扰方式估计 T_3 振幅 (参考 17.10 节; Bomble et al., 2005; Crawford and Stanton, 1998; Gwaltney and Head-Gordon, 2000; Gwaltney et al., 2000, 2002; Hirata et al., 2001; Kallay and Gauss, 2005; Kowalski and Fan, 2009; Kowalski and Piecuch, 2000; Kowalski and Valiev, 2009; Kucharski and Bartlett, 1998; Piecuch and Wloch, 2005; Piecuch et al., 2006; Raghavachari et al., 1989; Stanton, 1997; Taube and Bartlett, 2008a,b; Urban et al., 1985)。在这类形式中最流行的方法是 CCSD(T), 这种方法中的基态能量被 CCSD 能量 (E^{CCSD}) 的总和以及非迭代的 CCSD(T) 修正所表达, 这个修正结合了其中包含三激发中间态的标准多体问题扰动理论的展开式第四和第五个元素:

$$E^{\text{CCSD(T)}} = E^{\text{CCSD}} + \langle \Phi | (T_2^+ V_N) R_3^{(0)} V T_2 | \Phi \rangle + \langle \Phi | (T_1^+ V_N) R_3^{(0)} V T_2 | \Phi \rangle \quad (17-15)$$

其中, V_N 是电子汉密尔顿函数在常规乘积形式下的二体部分。CCSD(T) 方法是当前最常使用的方法。特别是在光学性质、几何优化和化学反应领域的研究中, 这个方法最为常用。

使用三体方法 $R_3^{(0)}$ 的定义, CCSD(T) 能量可以重写为:

$$E^{\text{CCSD(T)}} = E^{\text{CCSD}} + \sum_{\substack{i < j < k \\ a < b < c}} \frac{\langle \Phi | (T_2^+ V_N) | \Phi_{ijk}^{abc} \rangle \langle \Phi_{ijk}^{abc} | V T_2 | \Phi \rangle}{\epsilon_i + \epsilon_j + \epsilon_k - \epsilon_a - \epsilon_b - \epsilon_c} + \sum_{\substack{i < j < k \\ a < b < c}} \frac{\langle \Phi | (T_1^+ V_N) | \Phi_{ijk}^{abc} \rangle \langle \Phi_{ijk}^{abc} | V T_2 | \Phi \rangle}{\epsilon_i + \epsilon_j + \epsilon_k - \epsilon_a - \epsilon_b - \epsilon_c} \quad (17-16)$$

其中, ϵ 是轨道能量。CCSD (T) 修正中最消耗计算资源的部分 (有 $n_o^3 n_u^3$ 的缩放比例) 与是否存在 $\langle \Phi_{ijk}^{abc} | V T_2 | \Phi \rangle$ 项是相关的, 而这个表达式是如下定义的:

$$\begin{aligned} \langle \Phi_{ijk}^{abc} | V_N T_2 | \Phi \rangle = & v_{ma}^{ij} t_{bc}^{mk} - v_{mb}^{ij} t_{ac}^{mk} + v_{mc}^{ij} t_{ab}^{mk} \\ & - v_{ma}^{ik} t_{bc}^{mj} + v_{mb}^{ik} t_{ac}^{mj} - v_{mc}^{ik} t_{ab}^{mj} \\ & + v_{ma}^{jk} t_{bc}^{mi} - v_{mb}^{jk} t_{ac}^{mi} + v_{mc}^{jk} t_{ab}^{mi} \\ & - v_{ab}^{ei} t_{ec}^{jk} + v_{ac}^{ei} t_{eb}^{jk} - v_{bc}^{ei} t_{ea}^{jk} \\ & + v_{ab}^{ej} t_{ec}^{ik} - v_{ac}^{ej} t_{eb}^{ik} + v_{bc}^{ej} t_{ea}^{ik} \\ & - v_{ab}^{ek} t_{ec}^{ij} + v_{ac}^{ek} t_{eb}^{ij} - v_{bc}^{ek} t_{ea}^{ij} \\ & (i < j < k, a < b < c) \end{aligned} \quad (17-17)$$

其中, v_{rs}^{pq} 是二电子积分张量。式 (17-17) 能够分成收缩、被占据的下标 (A_{ijk}^{abc} ; (式 17-17) 中右边 (r.h.s) 的前 9 个变量), 以及对应于收缩、被占据下标的项 (B_{ijk}^{abc} ; (式 17-17) 右边的余下项):

$$\langle \Phi_{ijk}^{abc} | V_N T_2 | \Phi \rangle = A_{ijk}^{abc} + B_{ijk}^{abc} \quad (17-18)$$

TCE CCSD(T) 的实现使用了所有张量定义簇和汉密尔顿函数操作符的旋转轨道表达方式。这保证了 TCE CCSD(T) 方法可以使用不同参考函数, 包括闭合壳系统的限制性 Hartree-Fock 行列式以及限制性开壳和开壳分子的非限制性 HF Slater 行列式。然而, 相比于其他仅针对闭壳系统描述的 CCSD(T) 实现 (比如基于轨道非正交旋转适应形式的 NOSA-CCSD (T) 的 CCSD (T) 实现), 这个方法可以解决大量的簇振幅问题。

一般的 CCSD(T) 旋转轨道公式的求解比 NOSA-CCSD(T) 形式要慢, 原因有两点。首先, 在旋转轨道公式中有 4 种类型的 (T) 项, 中间的三激发的旋转结构是相关的: $|\Phi_{aaa}^{aaa}\rangle$, $|\Phi_{aab}^{aab}\rangle$, $|\Phi_{abb}^{abb}\rangle$, $|\Phi_{bbb}^{bbb}\rangle$ 。其次, 内部收缩是通过旋转轨道下标表现的 (相比于 NOSA-CCSD (T) 方法, 这个方法要花费两倍的代价)。事实上, 在旋转轨道中三激发振幅的反对称化会引入其他因素, 这些因素会导致旋转轨道 CCSD(T) 的计算复杂度上升。然而, 所有的因素一定会和基础理论的代数结构相关并且与给定实现的效率无关 (意味着缺失这些因素也不会影响什么)。

17.3 NWChem 软件架构

1993 年, 有两个主要的概念影响了 NWChem 的设计。

- 在大规模并行集群上的扩展性。
- 为了实现新的理论方法而采用的以模块为基础的架构。

这两种具有决定性意义的特征使得 NWChem 的发展显著区别于其他传统且成功地应用在计算化学领域的工具。

代码架构的模块化是通过 Fortran 的面向对象方法实现的。我们提供给读者的出版物中有对于这个主题的阐述, 这些出版物会细致地描述 NWChem 架构的不同部件。

17.3.1 全局数组

为了完成并行可扩展性, NWChem 采用全局数组 (GA) 工具包进行通信 (参考图 17-1)。GA 工具包是一个库, 它用来并行化其主要部分是大且密的数组的代码。GA 通过在并行计算机的分布式内存上分布数组并提供一套简易操作这些数组的方法, 为科研程序员提供了一个抽象层。更传统的消息传递方法需要发送者、接收者同步执行任务 (如数组转换), 而 NWChem 代码使用单边的 GA 操作达到了这样的目的。GA 可以分为两类操作: 聚合和局部。聚合操作需要调动全部进程参与进来, 而局部操作可能只有单个进程独自参与。预取 (ga_get) 或者更新 (ga_put) 是最常用的局部操作。通常来说, 线性代数操作是聚合操作 (比如矩阵乘法、求特征值等)。GA 提供了 Fortran 和 C/C++ 的语言绑定。这意味着当使用 MPI 实现它自己的一些功能 (比如进程创建、一些集合操作) 时, 这个库会与 MPI 兼容。

在给定计算机架构上 GA 的效率严重依赖于聚合远程内存复制接口 (ARMCI) 库。AEMCI 通过映射到很多不同的通信渠道 (比如 MPI、OpenIB、Portals 等) 实现了 GA 的基本通信层。

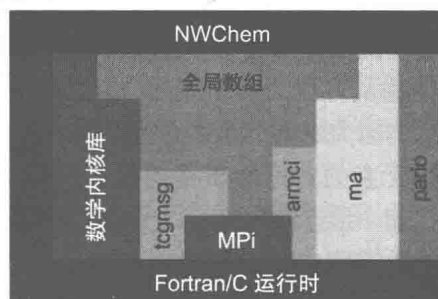


图 17-1 NWChem 软件栈的组件

17.3.2 张量收缩引擎

TCE 使用了一种多步方法自动求导和并行化量子多体方法, CC 就是一种可以使用的量子多体方法。首先, 把运算表达式转换成多维数组表达式。其次, 使用性能启发法把这些数组表达式转换为优化操作树。最后, 使用源模板实现优化操作树所表达的并行代码。因为 TCE 的出现, 第一种并行实现的数值方法比之前更广泛地使用到大规模科学问题上。原来 NWChem 中的 TCE 实现是泛泛意义上实现的 (即对于已知特例缺乏优化), 这种 TCE 要经过优化才能将 NWChem 中的 CC 方法拓展到超过 1000 个进程上。应用在 TCE 的优化手段基本涵盖了代码的所有方面, 包括更加紧凑的代码表示 (无旋转积分在标准案例中是立刻反对称的), 通过使用 TCE 编译器所不能实现的额外聚合来减少通信。

对于类似于以下的表达式:

$$Z(i, j, a, b) += \sum_{d, e} X(i, j, c, d) * Y(c, d, a, b) \quad (17-19)$$

这类表达式是 CCSD 等式求解的瓶颈, 这个表达式中, 在所有维度上的每个数组都参与了运算, 这些数据分布在 1 维 GA 中的机器上。这个应用并不使用多维 GA, 因为这里不能利用块稀疏性或者指数排列对称性。远程访问是通过使用一个针对数据块的查询表和一个 `ga_get` 操作得以实现的操作。但是全局数据层的结构对于本地计算并不总是合适的。因此, 在 `ga_get` 操作完成之后, 紧接着要将数据重新排列成方便计算的形式。为了标记的紧凑性, 我们使用 `Fetch` 操作来代表我们在数据重排之后刚刚提到的从全局内存复制数据到本地内存的操作。`Symm` 函数是一系列逻辑测试的压缩, 这些测试可判定一些特定的块是否非零。这些测试可视为块的索引, 但在块内忽视索引, 因为对每个块都进行分组, 这样所有基本元素的对称性质是相同的。每个块索引代表一组连续的索引, 所以收缩就发生在多维数组间, 而不是单个元素上。但是我们可以认为本地操作是两个块的点乘。

对于 (T) 模板的估值 (图 17-3) 是非常类似的。本地的 6D 三元中间态的计算所在的内层循环需要以类似于图 17-2 所示算法的方式预取 V 、 T_1 和 T_2 的贡献。

17.4 设计卸载解决方案

新的软件要在 Intel Xeon Phi 协处理器上运行时, 我们要重点考虑该使用何种操作模式和何种编程模式。在协处理器的本

```
Tiled Global Arrays: X, Y, Z
Local Buffers: x, y, z
for all i, j ∈ Otiles do
  for all a, b ∈ Vtiles do
    if Symm(i, j, a, b) == True then
      Allocate z for Z(i, j, a, b) tile
      for all c, d ∈ Vtiles do
        if Symm(i, j, c, d) == True then
          if Symm(a, b, c, d) == True then
            Fetch X(i, j, c, d) into x
            Fetch Y(c, d, a, b) into y
            Contract z(i, j, a, b) += x(i, j, c, d)*y(c, d, a, b)
          end if
        end if
      end for
      Accumulate z into Z(i, j, a, b)
    end if
  end for
end for
```

图 17-2 式 (17-19) 的默认 TCE 实现的伪代码

```
Tiled 4D Global Arrays: V, D
Local Scalar: es, ed
Local 4D Buffers: v, d
for all i, j, l ∈ Otiles do
  for all a, b, c ∈ Vtiles do
    if Symm(i, j, k, a, b, c) == True then
      Allocate ts, td (local 6D buffers)
      Compute ts ((Φabcijk|VNT1|Φ))
      Compute td ((Φabcijk|VNT2|Φ))
      Compute es (term 2 in Eq. 16)
      Compute ed (term 3 in Eq. 16)
    end if
  end for
end for
Reduce es, ed
```

图 17-3 式 (17-16) 的默认 TCE 实现的伪代码。ts 和 td 的计算包含 V (即 V_2) 和 D (即 T_2) 的块预取, 并执行与式 (17-17) 展示的安排相关的所有本地收缩

机模式或者对称模式下,协处理器都是自治的计算节点,这些节点在无论是否有其主机参与的情况下,通过运行 MPI 或者 GA 序列参与计算。而在卸载模式下,一部分计算仍在主机中完成,但一些内核连同它们的输入数据都卸载到协处理器中进一步处理。

我们认为 TCE CCSD(T) 算法的结构是非常适合使用卸载模型的。它包含了几个浮点计算密集和高度并行化的内核。代码中也有很多可以数据重用的代码段,主机和协处理器之间因此避免了大量的通信。除此之外,在本机模式中,一些算法的开始阶段(比如两电子排斥积分的估值)需要大量调试,然而在卸载模式下,我们能够在主机端进行这些计算。最后我们期望 GA 能够在主机端比协处理器端得到更高的消息率和在更高的通信带宽。

在本机模式、主机间的 MPI 实现的对称模式和卸载模式中,我们选了卸载模式。因此我们要分析代码中哪些部分适合卸载。在一些如同 NWChem 的复杂应用中,寻求一种合适的方法找到代码中具有卸载潜能的代码段并找到候选卸载部分会对卸载优化至关重要。这些方法对实现卸载并优化这些代码有所帮助。我们的方法是受自顶向下的迭代的软件优化方法所启发的(Yasin, 2014),这个方案通过热点分析发现应用代码中的瓶颈,从而优化软件。

图 17-4 是我们的方案的图形解释。我们先选择一个基准测试,这个基准测试尽可能地仿真应用,同时它的规模要小,可以控制。一个理想的基准测试运行时间大概不超过 15 分钟,更重要的是,能够触发和真实应用一样的代码区域。运行基准测试时,收集应用数据,并得到一个热点配置文件。在配置文件中的热点部分都是潜在的卸载选项。我们接下来做调用树分析,它将给我们关于(潜在的)普通函数的信息,这些函数能够用作卸载的常用定位点。下一步,我们需要鉴别潜在的热点代码是否适合卸载到协处理器上执行。接下来我们关注的信息(这可能需要额外的性能数据,比如,内存带宽、向量化潜力等)来自于循环分析。这里我们将检查一个热点并定位任何循环以及它们的最小计数、最大计数和平均循环计数。我们也加入了从编译器向量化报告中得到的更加复杂的循环信息,用于了解循环是否能够稳定地向量化和并行化。

最后三步接着处理了实际实现的工作。在分析之后,我们就可以加入卸载 pragmas 关键字(Intel 对卸载的拓展)、卸载的关键字(Intel Cilk Plus)或者 OpenMP 指令。在一段代码中加入卸载关键字有时候并不会得到最优解,因为这个关键字会引发 PCIe 总线上大量的数据传输。在这里,我们会结合调用树分析的结果提升卸载和数据传输的性能,并最小化数据传输。最后,我们优化单个卸载区域,从而提升这段代码在目标设备上的性能。

我们将上述方法应用到 NWChem CCSD(T) 方法的优化中。我们使用尿嘧啶二聚体作为基准测试,并使用 VTune Amplifier XE 分析热点。

图 17-5 展示了 GUI 的截屏以及应用的热点。我们能看到 comex_make_progress 函数消耗了 58% 的时间。这个函数的主要作用是处理 NWChem 各进程之间的通信。它耗时的原因是这个函数使用了很多循环来等待来自其他进程的消息。另外,这里有 38% 的计算时间来自于 18 个 sd_t_dx_y 函数(sd_t_d1_1 到 sd_t_d1_9 和 sd_t_d2 到 sd_t_d2_9)。它们都被另外一个锚函数(在 ccsd_t_double_1.F 中的 ccsd_d_doubles_1_2)调用。因为根据热点配置文件的信息,我们认为这些函数卸载到协处理器上会是一个不错的选择。

接着,分析调用树的信息,这个调用树文件显示了所有调用 sd_t_dx_y 函数的函数。图 17-6 展示了相应的 VTune Amplifier XE 的截屏。调用树分析显示所有的 sd_t_dx_y 都是被单个函数 ccsd_d_doubles_1_2 所调用的。就像我们在 17.5 节中将要看到的,这个函数将是我们主要的优化目标:我们要使它减少与协处理器设备来回的数据传输。

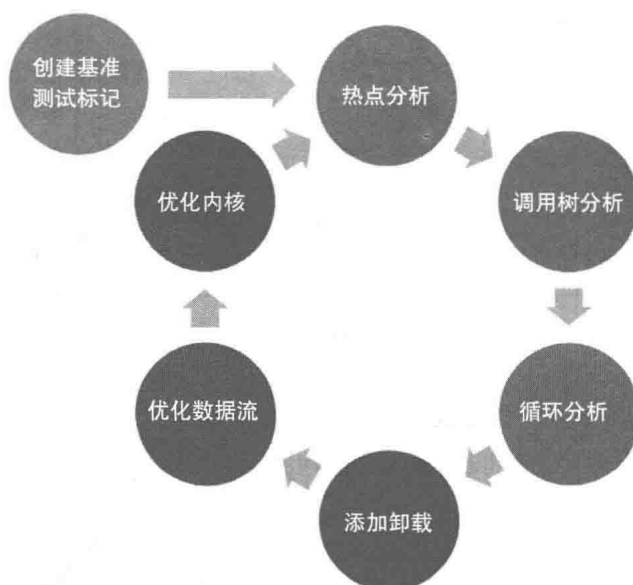


图 17-4 应用到 NWChem 的迭代分析方法

Function / Call Stack	CPU Time by Util...	Overhead and Spin Time	Module	Start Address
bcomex_make_progress	52.0%	0.250s	nwchem	0x295658
*sd_t_d2_2	4.2%	0s	nwchem	0x16a0d7
*sd_t_d2_8	4.2%	0s	nwchem	0x16a272
*sd_t_d2_9	4.2%	0s	nwchem	0x16a2e3
*sd_t_d2_5	4.2%	0s	nwchem	0x16a1a4
*sd_t_d2_3	4.1%	0s	nwchem	0x16a3a6
*sd_t_d2_6	4.1%	0s	nwchem	0x16a345
*sd_t_d2_1	3.1%	0s	nwchem	0x16a40b
*sd_t_d2_4	3.0%	0s	nwchem	0x16a142
*sd_t_d2_7	1.4%	0s	nwchem	0x16a211
p_mq_test	1.2%	0.030s	nwchem	0x29565a
Selected 18 row(s):		38.0%	0s	

图 17-5 Vtune Amplifier XE 对 NWChem CCSD (T) 方法的热点分析结果

Function Stack	CPU Time: Total by Utilization	Idle
ccsd_t_doubles_l_2	1333.239s	0
p_get_hash_block_i	335.529s	
p_get_hash_block	289.569s	
p_get_block	289.569s	
p_ga_get_	289.559s	
p_util_wallsec	0.010s	0
sd_t_d2_2	92.330s	92
sd_t_d2_8	91.870s	91
sd_t_d2_9	91.570s	91
sd_t_d2_5	90.490s	90
sd_t_d2_3	89.930s	89
sd_t_d2_6	88.020s	88
Selected 1 row(s):		1333.239s

图 17-6 图 17-5 中热点的调用树分析

下一步，我们将分析每个 `sd_t_dX_Y` 内核函数并找到适合卸载到协处理器的代码段。

多线程和 SIMD 向量化是实现协处理器高性能的关键。因此，我们需要评估这些内核是否适合做 SIMD 和多线程优化，进而决定哪些内核适合卸载到协处理器上。作为 18 个内核的代表，图 17-7 展示了 `sd_t_d1_1` 内核的代码，这些内核有相同的基本内核结构。每个内核由 7 个完美嵌套的循环组成，这些循环用来计算一个非常紧密的最内层的循环体。这些循环的计数在 20~30 之间，如此小的循环计数并不适合做 OpenMP 多线程优化。琐碎的（自动）向量化也会让代码的向量化潜能降低（大概只有 80%）。然而，这些循环并没有包含任何循环依赖（这些依赖会阻止并行化和向量化），因此这些循环很适合卸载到协处理器上。17.6 节将会展示如何优化才能提升向量化和并行化的潜能。

```

subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1             h7d,triplexx,t2sub,v2sub)
integer h3d,h2d,h1d,p6d,p5d,p4d,h7d,h3,h2,h1,p6,p5,p4,h7
double precision triplexx(h3d,h2d,h1d,p6d,p5d,p4d)
double precision t2sub(h7d,p4d,p5d,h1d), v2sub(h3d,h2d,p6d,h7d)
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
do h2=1,h2d
do h3=1,h3d
    triplexx(h3,h2,h1,p6,p5,p4) = triplexx(h3,h2,h1,p6,p5,p4)
c    - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
...
enddo
end

```

图 17-7 NWChem 的内核样例 (`sd_t_d1_1`)

有了分析得到的结论，我们就明白了如何将 NWChem 的 CCSD (T) 迁移到 Intel Xeon Phi 协处理器上。卸载到协处理器上的内核函数是 `sd_t_dX_Y`，这个函数在代码上是完全计算密集的片段。`ccsd_d_doubles_1_2` 函数是所有内核调用的普通定位点，同时我们将给 Fortran 代码引入卸载指令，从而使得协处理器处理内核函数并且优化定位点中的数据传输。

17.5 卸载架构

Intel Xeon Phi 协处理器支持多种卸载编程模型，每种都有自己的特点，只有 Intel LEO (Language Extension for Offloading) ——尽管在 OpenMP 4.0 中的 `target` 结构可用之前一种专用卸载语言——提供了本章优化所需要的数据传输和控制的灵活性，并且这种拓展对于现存 Fortran 源代码只做增量修改。

我们所采用的最重要的 LEO 指令将在图 17-8 中展示，这些指令用来在主机和协处理器间转移数据。

该算法的主要步骤和卸载解决方案的主要步骤如下。

1. `triplexx` 6 维数组 (式 (17-18))

```

#define ALLOC alloc_if(.true.) free_if(.false.)
#define FREE  alloc_if(.false.) free_if(.true.)
#define REUSE alloc_if(.false.) free_if(.false.)

cdir$ offload target(mic:mic_device)
N      in(triplexx:length(0) REUSE)
I      in(h3d,h2d,h1d)
I      in(p6d,p5d,p4d,h7d)
I      in(t2sub:length(1_t2sub) REUSE)
R      in(v2sub:length(1_v2sub) REUSE)

```

图 17-8 用于传输数据和实现从主机到协处理器的控制的卸载指令

中的 $\langle \Phi_{ijk}^{abc} | V_N T_2 | \Phi \rangle$ 首先在协处理器上创建并初始化为 0 (因此此处不需要数据转移)。

2. 接下来, 如图 17-8 所示, 在调用每个 `sd_t_dX_Y` 内核函数前, 都要加上卸载指令。其中, 每个 `sd_t_dX_Y` 内核函数用于将 `v2sub` 和 `t2sub` 这两个四维数组相乘。而对于 `triplesx` 数组, 现阶段没必要传输这个数组的数据。在卸载阶段, 我们直接在协处理器上累加 `v2sub` 和 `t2sub` 相乘的结果。传输两个四维数组是仅有的数据传输。它们首先在主机上计算然后复制到协处理器上 (鉴于数组的大小要求在小于 1 微秒内完成)。我们使用同步卸载模式: 主机线程和内核在等待从协处理器返回数据的时候处于闲置状态。

3. 当主机内核闲置并等待从协处理器返回结果时, 我们在相同的节点上使用空闲的内核执行另一个 GA 进程的线程。使用这个方法, 我们能够重用 GA 现存的负载均衡的设施。这保证了协处理器上的进程, 同主机 CPU 之间的负载不均衡是自动补偿的。异步卸载将需要把现存的负载均衡设施扩展为将异步负载均衡考虑进去的两阶段算法。在 17.7 节我们会看到这方面的更多内容。

4. 一旦所有结果计算出来, 在每个任务结束的时候大的 6 维 `triplesx` 数组就从协处理器一次性复制到主机端。在每个任务中, 一个特定内核执行的次数正比于占据的 `noab` 块总数 (式 (17-18) 中的 A 项) 或者未占据的 `nvab` 块总数 (式 (17-18) 中的 B 项)。

图 17-9 总结了上述数据管理和卸载步骤。为了简洁, 我们只用伪代码展示了代码的结构并且使用 `sd_t_dX_Y` 作为 18 个计算内核的代表。

```

cdir$ offload_transfer target(mic:dev) nocopy(triplesx:length(tsx_l) ALLOC)
cdir$ offload_transfer target(mic:dev) nocopy(t2sub:length(t2sub_l) ALLOC)
cdir$ offload_transfer target(mic:dev) nocopy(v2sub:length(v2sub_l) ALLOC)
cdir$ offload target(mic:dev) nocopy(triplesx:length(0) REUSE)
      call zero_triplesx(triplesx)
      do ...
        if (...)
cdir$ offload target(mic:dev)
      N      in(triplesx:length(0) REUSE)
      I      in(h3d,h2d,h1d)
      I      in(p6d,p5d,p4d,h7d)
      I      in(t2sub:length(l_t2sub) REUSE)
      R      in(v2sub:length(l_v2sub) REUSE)
          call sd_t_dX_Y(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
        endif
      enddo

cdir$ offload_transfer target(mic:dev) out(triplesx:length(tsx_l) REUSE)

```

图 17-9 `ccsd_d_doubles_1_2` 函数使用卸载数据管理和转移控制指令的伪代码, 使用 `sd_t_dX_Y` 作为 18 个要调用的内核的示例

17.6 内核优化

如 17.4 节的循环分析展示, 使用原始代码实现的线程分配和向量化操作 `sd_t_dX_Y` 内核函数无法充分利用协处理器。因此, 有必要对代码做一些改进。本节将展示如何优化 Fortran 代码, 这些改变将提升每个计算内核的线程效率和向量化能力。因为这些内核优化方法具有相似性, 所以只讨论一部分示例内核的情况。就像先前提到的, 我们的目标是保证 Fortran

代码尽可能不修改,同时如果修改代码,优化必须同时提升协处理器和主机的执行效率。

代码优化也是一项需要效率的工作,我们不会用底层编程模型(比如 C/C++)去重新写代码。除了修改 Fortran 代码之外,我们将完全依赖 Fortran 的 Intel Composer XE 2013 和它对 OpenMP 的实现来实现自动向量化。

图 17-10 展示了我们将要应用到 OpenMP 指令,这些指令要加入到图 17-7 的内核中。因为外层循环的循环计数少(计数大概只有 20 ~ 30,具体情况依据输入块的大小而变化),对于 Intel Xeon Phi 协处理器来说,并不能提供足够的并行性。我们使用 collapse 子句来指引 OpenMP 编译器将多个外层循环融合成一个乘积循环。这个操作会使得每个循环的循环计数至少比原先的最外层循环多两个数量级。OMPCOLLAPSE 这个参数在 NWChem 编译过程中可以用来调整循环展开的情况。对于绝大多数输入集,我们的试验已经展示了这个参数的最佳估值是 3。这意味着把 p4-p6 循环融合为单个长时间运行的乘积对于提升效率是最有效的。

```

subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1             h7d,triplex,t2sub,v2sub)
integer h3d,h2d,h1d,p6d,p5d,p4d,h7d,h3,h2,h1,p6,p5,p4,h7
double precision triplex(h3d*h2d,h1d,p6d,p5d,p4d)
double precision t2sub(h7d,p4d,p5d,h1d), v2sub(h3d*h2d,p6d,h7d)
!$omp parallel do private(p4,p5,p6,h2,h1,h3,h7) collapse(OMPCOLLAPSE)
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h7=1,h7d
!dec$ loop count max=1000, min=20
do h3h2=1,h2d*h3d
triplex(h3h2,h1,p6,p5,p4) = triplex(h3h2,h1,p6,p5,p4)
- t2sub(h7,p4,p5,h1)*v2sub(h3h2,p6,h7)
...
enddo
!$omp end parallel do
end

```

图 17-10 使用 OpenMP 和手动循环融合的图 17-7 所示示例内核

对于 Intel Composer XE 来说,内存循环是非常适合做自动向量化的。它既不包含影响循环的依赖性(这会阻止向量化),也没有出现非一致步长的情况。尽管看上去非常完美,但向量化内层循环的结果并不高效。向量化潜力表明了一个向量寄存器的平均填充程度和向量化效率。越高的向量化潜力就意味着寄存器中的更多向量通道在计算中被使用。在最优的情况下,向量化潜力接近 100%,这就意味着向量指令在向量寄存器中全程满负荷工作。

在该示例中,向量化潜力在块大小为 20 的时候大概为 83%。因为块大小通常不是向量宽度的倍数,所以编译器需要发出一个剩余的循环,这个循环用来考虑多余的循环迭代。对于一个向量宽度是 8 个双精度元素和一个大小为 20 的块,向量化的循环将会处理 16 个元素(两遍全向量宽度),剩余循环执行 4 次迭代。一般来说,这导致了该向量和剩余循环的向量化潜力是每向量 6.6 个元素或者说 83% 的使用率。

手动融合两个最内层循环(h3 和 h2 融合为一个 h3h2 循环)使循环潜力接近 100%。因为手动融合循环后,循环的计数提升了最少一个数量级。对于上述大小为 20 的块的情况

来说, 向量化潜力也达到了最优。h2h3 循环要运行 20×200 (400) 次迭代, 400 是 8 的倍数, 向量化潜力因此得到了提升。融合内层循环也带来了额外的性能提升, 因为一些循环开销与两个 (短) 嵌套循环相关, 现在我们有效避免了乘积循环并减少了开销。

除了代码变换之外, 我们也使用 Intel Composer XE for Fortran 的编译指令。这些指令如图 17-10 所示。这些指令告诉编译器期望的最小和最大的内层嵌套循环的计数。通过 17.4 节的循环分析获得这些信息。没有指令, 编译器无法了解循环计数, 并且会过高估计实际执行的迭代次数。对于一些内核来说, 这往往会使得编译器优化起到反效果 (比如, 不正确的循环阻塞或者循环展开), 这些对协处理器来说会阻碍最优向量化。避免这些自动编译器转换 (通过增加编译指令) 能大致提升 10% 的性能。

在有 18 个 `sd_t_dx_Y` 函数的集合中, 一些内核的向量化并不尽如上述方法那样简单易行。图 17-11 展示了其中一个出问题的内核。如果这个内核按照上述代码进行编译, 编译器为了访问 `t2sub` 将会发出、聚集并且扩散指令。实际上要提升这类循环嵌套的效率并不容易。循环 `h2, h7, h3` 和 `h2` 的任何排列至少会对一个数组的访问引入聚集 / 扩散指令。然而, 如果 `t2sub` 被转置, 就能避免聚集 / 扩散指令, 而且这使用单位步长的读取操作能够很好地向量化。这些转换可以提升类似图 17-11 中内核的效率, 并且这个转换并没有影响其他内核的运行。访问 `t2sub` 必须使用这样的循环, 同时转置 `t2sub` 对内核的其他计算来说没有任何不同。这个转置操作是紧接着从其他进程接收到数据之后在主机进程上完成的, 相比于内核在目标设备上的运行时间来说, 这点时间消耗微不足道。事实上, 这个开销将完全掩盖在系统噪声中。

```

subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1              h7d,triplesx,t2sub,v2sub)
integer h3d,h2d,h1d,p6d,p5d,p4d,h7d,h3,h2,h1,p6,p5,p4,h7
double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
double precision t2sub(h7d,p4d,p5d,h1d), v2sub(h3d*h2d,p6d,h7d)
!$omp parallel do private(p4,p5,p6,h2,h1,h3,h7) collapse(OMPCOLLAPSE)
do p4=1,p4d
do p5=1,p5d
do p6=1,p6d
do h2=1,h2d
do h3=1,h3d
do h7=1,h7d
do h1=1,h1d
    triplesx(h1,h3,h2,p6,p5,p4) = triplesx(h1,h3,h2,p6,p5,p4)
c      - t2sub(h7,p4,p5,h1)*v2sub(h3,h2,p6,h7)
...
enddo
!$omp end parallel do
end

```

图 17-11 使用非一致步长内存访问的 NWChem 的 `sd_t_d1_1` 内核

17.7 性能评估

我们实现的 NWChem 可以高效并行运行在由 460 个节点组成的集群上。这个集群由双插槽 8 核 Intel Xeon E5-2670 2.6GHz 处理器和 128GB 内存所构成。每个节点有两个 60 核 Intel Xeon Phi 5110P 1.053MHz 协处理器和 8GB 的片上 GDDR5 内存。这意味着共有 7360

个 Xeon 处理器内核和 62 560 个异构内核可以用于提供卸载计算。作为基准测试, 我们使用闭壳 (closed shell) 并五苯分子 ($C_{22}H_{14}$) 上计算 CCSD(T) 相关能量的矫正。这个基准测试的块大小是 24, 这不超过 8GB 片上可用内存的限制。

从更专业的化学角度讲, 输入数据对于 378 个基础函数来说使用了一个 cc-pVDZ 基础集合, 使用了 51 个相关的占据的轨道 (在一个冷冻核近似中), 以及 305 个未占据的相关轨道。测试计算中没有使用对称性。

对于集群来说, 最好的异构启动是每个节点运行 8 个 GA 进程。这 8 个进程中的 4 个在主机上执行并产生 4 个线程, 这样就能够充分利用所有 16 个主机内核。其余的 4 个进程并发运行在协处理器设备上。因为所有的卸载是异步操作, 所以主机进程将会在等待卸载结果时运行其他计算。其他 (仅在主机上的) 进程能够借助其 OpenMP 线程使用空闲内核。这就是为什么我们可以在主机系统上超额使用 16 个 OpenMP 线程。

受到第 12 章的启发, 我们并发地在多主机进程向相同设备启动卸载。第一个进程分配给第一个目标设备的第一部分 (第 0 ~ 28 个内核), 第二个进程分配给第一个目标设备的第二部分 (第 28 ~ 57 个内核)。它们都在一个物理内核上执行 4 个超线程, 这就对应了 116 个 OpenMP 线程。通常我们留两个空闲内核给操作系统, 以及用来处理数据转移。相对应地, 第三和第四个进程执行第二个协处理器的第一和第二部分。

在协处理器中为每个卸载分配了相应的内存: `triplesx` 数组需要占大约 1.5GB, 而每个 `t2sub` 和 `v2sub` 数组占大约 2.7MB。如果我们使用协处理器上的大页, 那么就可以减少在数据传输和计算时发生的页错误次数, 从而减少大概 30% 的数据传输时间。我们也减少了因为协处理器上的 TLB 未命中而带来的额外损失。对于每个卸载线程, 使用 `MIC_USE_2MB_BUFFERS` 环境变量设定 16KB 的大页。这个值设定了大页上可以存储 16KB 以上的数据, 这在本例中是足够直接传输到内存大页上的。

如图 17-12 所示, 异构卸载方法 (“Xeon” & “Xeon Phi”) 相比纯粹的主机架构 (仅有 “Xeon”) 有大约 2.5 倍的加速比。我们的实现可以让全部 460 个节点上的协处理器保持性能优势。我们也测试了单个协处理器配置下的性能。我们忽略了主机上的计算量并只用主机进程作为通信桩 (它们仅用于消息传递并立即转换给协处理器)。因此, 相对于异构实现, 单个协处理器的配置会使性能大约下降 0.77 ~ 0.90。这证明真实的异构计算能够有效地使用全部的 Xeon 和 Xeon Phi 计算资源。

从第二个扩展的基准测试可以看到, 一个 1,3,4,5-tetrasily-limidazol-2-ylidene 分子 (化学式为 $Si_4C_3N_2H_{12}$) 处于三元组状态。基准测试建立起来, 这样对于全部 706 个基础函数可以使用一个 aug-cc-pVTZ 基础集, 在冷冻的核近似中使用 26 个相关的 α 被占据的轨道 (24 个相关的 β 被占据的轨道) 和 655 个未被占据的 α (657 个未被占据的 β) 轨道。这次, 我们在测试计算中也没有使用对称性 (图 17-13)。

第二个基准测试有更高的浮点性能要求, 因此使用卸载协处理器模式会有优异的加速比。在异构环境下, 结果比仅有主机的 TCE 节点高了 2.7 倍。

17.8 总结

在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器搭建的异构环境计算能够加快求解科学应用的速度。NWChem 化学包是一个将这些原则应用到实际的范例。比起传统的优化方法, Intel Xeon Phi 协处理器是一种更好的选择。使用相同的编程工具, Intel Xeon Phi 协处理器

的代码有较强的移植性并且编程模型和以前也相同。我们通过卸载关键的内核到协处理器上就可以达到目的。

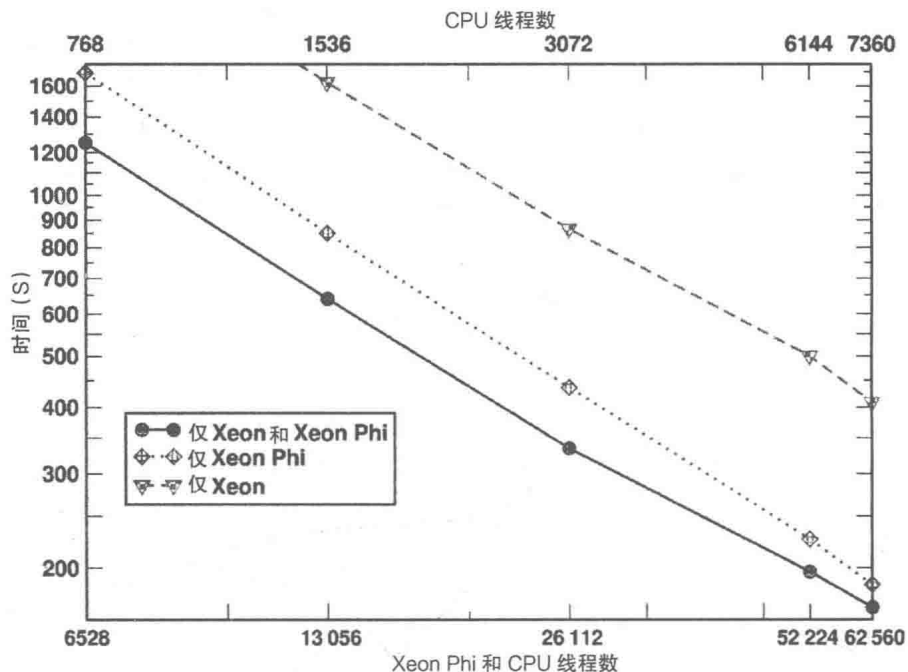


图 17-12 并五苯分子 ($C_{22}H_{12}$) 对 CCSD (T) 相关能量的三元组扰动修正 (在 s 中) 的时间。所有坐标轴的数字都取对数

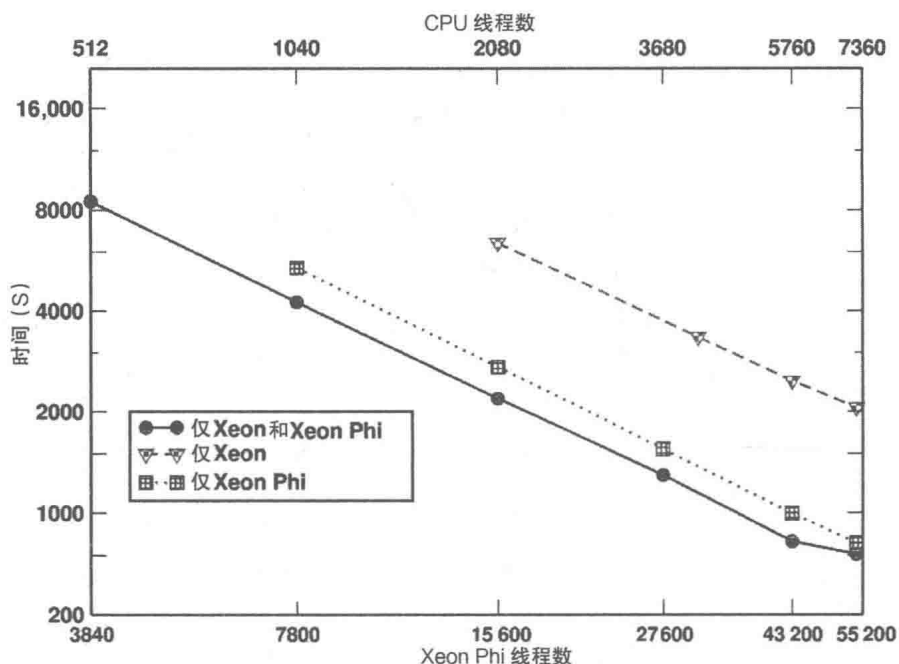


图 17-13 1,3,4,5-tetrasilylimidazol-2-ylidene 分子 ($Si_4C_3N_2H_{12}$) 在 三元组 状态下对 CCSD(T) 相关能量的三元组扰动修正 (在 s 中) 的时间。所有坐标轴的数字都取对数

我们对 NWChem CCSD(T) 的异构卸载解决方案展示了在大规模集群上不但可能获得高性能而且具有可行性, 该方案依赖于标准编程语言 (Fortran) 和并行编程方法 (OpenMP)。这些建立的块很适合将需要的内核以最高速度卸载到协处理器上并且不损失工具链的生产效率。在主机和协处理器上使用相同的编程模型也能优化进程, 因为所有对协处理器做的改进也能立即在主机端提升性能。

决定哪些部分用作卸载的分析方法, 可以得到一个简单的移植过程。使用这个分析步骤发现热点, 发现可以用于卸载的固定点并分析循环, 循环分析可以为代码优化提供有价值的信息。接着使用这些信息去决定代码中哪部分要被卸载到协处理器上。分析它们的调用树从而发现数据和卸载转换的最佳位置。最后分析计算内核的结构并提升它们的向量化和并行化潜力。

借助 Intel Xeon Phi 协处理器额外的计算部件并使它加速可得到 NWChem 的高精度 CCSD(T) 方法, 并且有助于实现更精确的仿真 (进一步讨论参见 Apra et al., 2014)。并设有使不精准系统中算法更简单的仿真范围变窄 (比如针对闭壳分子系统的 Kobayashi-Rendell 方法), CCSD(T) 方法可以广泛应用到不同的分子系统类型中 (开壳未配对的电子)。因为更高的仿真精度会产生更高的计算强度。应用不能只仿真小分子系统, 也不应有较大的时间开销。卸载的 NWChem CCSD(T) 方法大致提升了 3 倍的性能, 这些提升弥补了 NWChem 计算上的缺陷, 让它可以进行高精度仿真并减少求解时间。

Intel Xeon Phi 协处理器上的工作应该可以迁移到下一代 Intel Xeon Phi 产品上 (代号为 Knights Landing (KNL)), 这代芯片预计可以作为基于套接字的解决方案, 这使得不再需要卸载内核到协处理器上, 并提供了一些高带宽的内置存储器。KNL 的单线程性能预计是今天的 Intel Xeon Phi 协处理器的三倍。NWChem 的 CCSD(T) 方法的其他部分可以在众核芯片上处理, 不需要使用协处理器。我们已经实现的基于协处理器的 OpenMP 并行内核可以作为一个起点, 用这些代码可以逐渐完成多线程的内核。我们预计, 一个真正的混合 GA / OpenMP 实现将需要充分利用 KNL 处理器的性能。提高剩余代码库的向量化潜力也是一个需要做的工作。该内置存储器应该是一个有趣的优化目标, 优化它可以提升 NWChem 内存访问的局部性。

17.9 致谢

这里描述的研究成果是 EMSL 完成的, EMSL 是科学用户设施 DOE 办公室, 位于太平洋西北国家实验室, 由生物和环境研究办公室赞助。

17.10 更多信息

NWChem 6.5 版本的代码和额外信息可以从 <http://lotsofcores.com/NWChem> 获得。

- Aprà, E., Klemm, M., Kowalski, K., November 2014. Efficient implementation of many-body quantum chemical methods on the Intel Xeon Phi coprocessor. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, New Orleans, LA.
- Bomble, Y.J., Stanton, J.F., Kallà, M., Gauss, J., 2005. Coupled-cluster methods including noniterative corrections for quadruple excitations. J. Chem. Phys. 123 (5), 054101.

- Crawford, T.D., Stanton, J.F., 1998. Investigation of an asymmetric triple-excitation correction for coupled-cluster energies. *Int. J. Quantum Chem* 70 (4-5), 601-611. International Symposium on Atomic, Molecular, and Condensed Matter Theory at the 38th Annual Sanibel Symposium, St. Augustine, FL, February 21-27, 1998.
- Gwaltney, S.R., Byrd, E.F.C., Van Voorhis, T., Head-Gordon, M., 2002. A perturbative correction to the quadratic coupled-cluster doubles method for higher excitations. *Chem. Phys. Lett.* 353 (5-6), 359-367.
- Gwaltney, S.R., Head-Gordon, M., 2000. A second-order correction to singles and doubles coupled-cluster methods based on a perturbative expansion of a similarity-transformed Hamiltonian. *Chem. Phys. Lett.* 323(1-2), 21-28.
- Gwaltney, S.R., Sherrill, C.D., Head-Gordon, M., Krylov, A.I., 2000. Second-order perturbation corrections to singles and doubles coupled-cluster methods: General theory and application to the valence optimized doubles model. *J. Chem. Phys.* 113 (9), 3548-3560.
- Hirata, S., Nooijen, M., Grabowski, I., Bartlett, R.J., 2001. Perturbative corrections to coupled-cluster and equation-of-motion coupled-cluster energies: a determinantal analysis. *J. Chem. Phys.* 114 (9), 3919-3928.
- Kucharski, S.A., Bartlett, R.J., 1998. Noniterative energy corrections through fifth-order to the coupled cluster singles and doubles method. *J. Chem. Phys.* 108 (13), 5243-5254.
- Kowalski, K., Fan, P.-D., 2009. Generating functionals based formulation of the method of moments of coupled cluster equations. *J. Chem. Phys.* 130 (8), 084112.
- Kall  y, M., Gauss, J., 2005. Approximate treatment of higher excitations in coupled-cluster theory. *J. Chem. Phys.* 123 (21), 214105.
- Kowalski, K., Piecuch, P., 2000. The method of moments of coupled-cluster equations and the renormalized CCSD[T], CCSD(T), CCSD(TQ), and CCSDT(Q) approaches. *J. Chem. Phys.* 113 (1), 18-35.
- Kowalski, K., Valiev, M., 2009. Extensive regularization of the coupled cluster methods based on the generating functional formalism: application to gas-phase benchmarks and to the S(N)2 reaction of CHCl3 and OH-in water. *J. Chem. Phys.* 131 (23), 234107.
- Piecuch, P., Wloch, M., 2005. Renormalized coupled-cluster methods exploiting left eigenstates of the similarity transformed Hamiltonian. *J. Chem. Phys.* 123 (22).
- Piecuch, P., Wloch, M., Gour, J.R., Kinal, A., 2006. Single-reference, size-extensive, non-iterative coupled-cluster approaches to bond breaking and biradicals. *Chem. Phys. Lett.* 418 (4-6), 467-474.
- Raghavachari, K., Trucks, G.W., Pople, J.A., Head-Gordon, M., 1989. A 5th-order perturbation comparison of electron correlation theories. *Chem. Phys. Lett.* 157 (6), 479-483.
- Stanton, J.F., 1997. Why CCSD(T) works: a different perspective. *Chem. Phys. Lett.* 281 (1-3), 130-134.
- Taube, A.G., Bartlett, R.J., 2008a. Improving upon CCSD(T): Lambda CCSD(T). I.

Potential energy surfaces. J. Chem. Phys. 128 (4), 044110.

- Taube, A.G., Bartlett, R.J., 2008b. Improving upon CCSD(T): Lambda CCSD(T). II. Stationary formulation and derivatives. J. Chem. Phys. 128 (4), 044111.
- Urban, M., Noga, J., Cole, S.J., Bartlett, R.J., 1985. Towards a full CCSDT model for electron correlation. J. Chem. Phys. 83 (8), 4041-4046.
- Yasin, A., 2014. A top-down method for performance analysis and counters architecture. In: Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software. Monterey, CA, pp. 35-44.

大规模多系统上的高效嵌套并行

Evgeny Fikshan*, Anton Malakhov†

* 美国, Intel 公司, † 俄罗斯, Intel 公司

本章提出了一种在 Intel Xeon Phi 协处理器和 Intel Xeon 处理器上嵌套并行的方法, 这两种处理器都包含大量内核, 并使用非一致性内存访问 (NUMA)。目前的解决办法是基于 Intel 线程构建模块 (Intel TBB) 任务平台的。

18.1 动机

最近 Intel Xeon 处理器和 Intel Xeon Phi 协处理器在内核数量上的增加使我们开始重新审视并行库中所采取的方法。一般来说, 我们已经觉察到可扩展性已经成为一个问题, 原因是, 当处理的数据总量保持不变时, 如果有更多线程需要同步, 那么一个线程库的开销会趋于增长。其结果是, 整个内核的可扩展性都会受到影响。一些应用程序是由任务并行原则驱动的, 或者有并行地运行多个任务的能力, 并且也可以数据并行驱动 (在一个数据集中有并行运行单个任务的能力)。任务和数据并行的组合提供了一个可以最小化线程库开销的机会。

NUMA 鼓励使用一种方法来最小化内部节点间的存储传输, 即对于特定的节点本地化执行数据并行任务——同本章讨论的其他方法一起使用。现代的 Intel Xeon 处理器的服务器系统通常有两个或多个 NUMA 节点 (CPU 插槽)。每一个节点都有其自己的一套存储体, 并由多个处理器 (内核) 组成。一个节点中独立的处理器可以通过 Intel Quick Path Interconnect (Intel QPI), 总线访问另一个节点中的存储体。然而, 这样的访问会导致通信延迟。但是, 线程库 (如 Intel TBB), 对于在以最有效的方式匹配用户的数据布局与 NUMA 拓扑结构这个问题上, 并没有什么帮助。任务可以在整个系统里扩展以充分并行开发, 把它们映射在一个 NUMA 节点的范围內, 可以使之更有效率。

这种方法已经应用于 STAC A2 基准测试, 并且已经通过简单地使用线程或者多任务库使得性能大幅度提升。

18.2 基准测试

为了本章的目的而开发出来的一个人工基准测试, 是用来测试 Intel Xeon Phi 协处理器上线程库的开销和实体存储器局限性的, 并测量 NUMA 对多插槽 Intel Xeon 平台的影响。基准测试的内核任务计算一个双精度浮点数组的平方根。被选中数组的尺寸足够大以超过处理器最后一级缓存 (LLC) 的大小, 从而迫使产生 DRAM 存储访问并增大 NUMA 的开销。每个内核任务实例, 通过一个工作线程执行, 并被配置成可以持续足够长的时间, 以模拟真正工作负载的计时。OpenMP 4.0 simd 指令用来生成在现代处理器上可用的向量指令。图 18-1 列出了基准测试的主循环。完整的基准测试源代码可在 <http://lotsofcores.com> 获取。

除了在一个内核任务内执行数据并行的并行循环之外, 一个内核任务的多个实例是同时

执行的, 因此呈现出额外的任务(或者功能)并行。这种执行方案产生多级并行, 这也称为嵌套并行, 由于并行循环被嵌套进多重并行执行的任务中。

```
#define DATA_SIZE (32*1024*1024)
#define BLOCK_SIZE (4*1024)
typedef std::vector<double,
                tbb::cache_aligned_allocator<double> >
                data_vector_t;
void compute_sqrt_block(const double* src, double* dst,
                        int begin, int end)
{
    #pragma omp simd aligned(src, dst: 64)
    for(int i=begin; i<end; ++i){
        dst[i] = sqrt(src[i]);
    }
}
void kernel_task() {
    data_vector_t src(DATA_SIZE);
    data_vector_t dst(DATA_SIZE);
    // Generate input data
    tbb::parallel_for(0, DATA_SIZE, BLOCK_SIZE,
        [&](int i) {
            auto rand_val = std::bind(distribution, generator);
            int begin = i,
                end = std::min(i+BLOCK_SIZE, DATA_SIZE);
            for( int j=begin; j<end; ++j)
                src[j] = rand_val();
        });
    // Perform multiple iterations on the same data block
    tbb::affinity_partitioner partitioner;
    for(int t=0; t<ITERATIONS; ++t) {
        tbb::parallel_for(0, DATA_SIZE, BLOCK_SIZE,
            [&](int i) {
                int begin = i,
                    end = std::min(i+BLOCK_SIZE, DATA_SIZE);
                #pragma noline
                compute_sqrt_block(&src[0], &dst[0], begin, end);
            }, partitioner);
    }
}
```

图 18-1 基准测试内核任务例程, 基于 Intel TBB 模型

18.3 基线基准测试

上面提到的基准测试的性能评估是在最新可用的 Intel 平台上执行的。系统由双插槽 Intel Xeon E5-2695 v2 处理器和 Intel Xeon Phi 7120A 协处理器组成。Intel Composer XE 2013 SP1 更新了 3 个编译器用来为 Intel Xeon 处理器和 Intel Xeon Phi 协处理器提供编译。使用三个不同的线程库 Intel TBB 4.2、OpenMP 4.0 和 Intel Cilk Plus 来评估。GCC 4.7 或更高版本在测试机上对于 C++11 特性的支持是有效的。

基准测试的目标是尽可能快地并行执行一数据池的任务。由于每个任务需要大量的内存用于执行, 因此同时运行的任务数量会被可用的实体存储器的数量所限制, 如在 Intel Xeon Phi 协处理器上。

图 18-2 根据处理的任务数量描述了线程库的效率, 与在 Intel Xeon Phi 7120A 协处理器上的测量方法相同。效率的最优值为 1, 定义如下:

$$E = \frac{T_{\text{Based}}^{\text{Normalized}}}{T_{\text{Measured}}^{\text{Normalized}}} = \frac{T_{\text{Task}}^{\text{Serial}} / N_{\text{cores}}}{T_{\text{Measured}} / M_{\text{tasks}}}$$

式中, $T_{\text{Task}}^{\text{Serial}} / N_{\text{cores}}$ 代表了在 N 个内核的机器上执行一个任务的最佳时间, T_{Measured} 是在相同机器上执行 M 个任务所需要的时间; 因此 $T_{\text{Measured}} / M_{\text{tasks}}$ 代表一个任务的有效执行时间。测量时间不包括线程库的初始化时间。

Intel TBB 和 Intel Cilk Plus 的效率值均接近 1。不过, 当存储资源达到临界点时, 效率峰值出现在临界点之后。每个线程库有不同的行为, 这取决于内部构件的调度程序。Intel TBB 的效率值在 $M_{\text{tasks}} = 20$ 之后大大降低, 但是在 Intel Xeon Phi 7120A 协处理器上最多能够处理 34 个任务, OpenMP 和 Intel Cilk + 只能分别处理 29 个和 32 个任务。OpenMP 对于这个基准测试的效率值非常低, 这同 OpenMP 处理嵌套并行的方式有关。

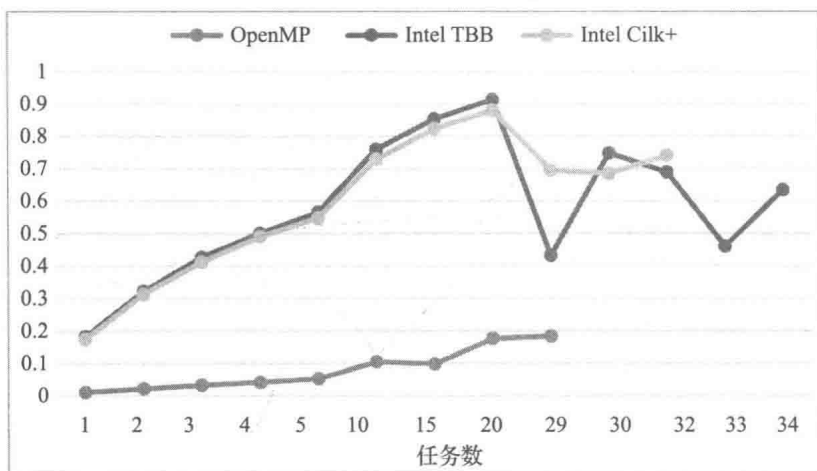


图 18-2 基准测试性能与内核任务数量的关系

没有一个线程库能够处理数量大于最大值的任务。克服 Intel Xeon Phi 协处理器的内存限制和缺乏交换空间的解决方法是限制同时并行执行的内核任务的数量。下一节介绍的两种方法通过使用 Intel TBB 的功能来帮助部署那些设备上的内存饥饿程序, 尤其是并行流水线 and 用户管理任务调度平台特征。Intel TBB 库并没有多少关于 C++ 类和多任务应用支持的设定。

18.4 流水线方法——Flat_arena 类

如上一节所示, 线程库同时高效执行的任务数量是特定的。此需求的立即执行依靠一个众所周知的同步原语信号量 (semaphore)。然而, 在一个工作线程内使用一个信号量会拖延该线程并且会有效地减少计算资源量。因此, 同步原语不能用来限制执行的任务数量。Intel TBB 库有并行流水线结构, 这可以用来限制执行任务的数量。

流水线的特性由类 `tbb::pipeline` 成员或者通过函数 `tbb::parallel_pipeline` 成员提供, 两者使得类型安全和 `lambda` 表达式有效。`tbb::parallel_pipeline` 用来创建一个由一系列 `filter` (流水线的阶段) 组成的流水线, 以用于一连串的题目。每个 `filter` 按下面三种模式中的一种来操作:

- 并行——无序地并行处理一个 `filter` 的多重实例。
- 有序串行——以相同的顺序一次处理一个 `filter` 的单个实例。

- 无序串行——无序地一次处理一个 filter 的单个实例。

流水线接口的其中一个参数是最大数量的活跃令牌数，在任何给定时间内该参数能控制流水线条目的数量。

图 18-3 展示了如何使用 `tbb::parallel_pipeline` 进行任务区域内的任务分派。

给出的 `parallel_pipeline` 实现方法要求任务列表要事先构建。不过，可以通过加强初步实现处理动态任务地提交，例如使用 `tbb::concurrent_queue`。

然而，为流水线地执行，转化程序结构可能会不太方便。在最外层，程序可能会通过 `parallel_for`、`flow::graph` 或者其他高级 TBB 算法组织，这可能会使得程序在实际中难以分成单独的任务并把它们人工地反馈给流水线，因此程序可能会失去使用这些结构的好处。

一个替代方法是基于多等级任务调度平台的，在下面几节中将会讲到，而且将会适合于任何类型的高级 TBB 算法。

```
parallel_pipeline( /*max number of live token=*/16,
    make_filter<void, std::function<void(void)>> >(
        filter::serial,
        [&](flow_control& fc)->auto{
            if( !list.empty() ) {
                std::function<void(void)> f = list.front();
                list.pop_front();
                return f;
            } else {
                fc.stop();
                return std::function<void(void)>([]{});
            }
        }) &
    make_filter< std::function<void(void)>,void>(
        filter::parallel,
        [](std::function<void(void)> f)->float
        { f(); }
    ));
```

图 18-3 使用 `tbb::parallel_pipeline` 在调度平台内进行任务分派

18.5 Intel TBB 用户管理任务调度平台

在 Intel TBB 术语中，一个任务调度平台是工作线程分享和抢断任务的地方。每个应用线程或者主线程，通过 Intel TBB 库产生的工作线程帮助维持其隐式的任务调度平台。简单来讲，工作线程初始在一个全局线程池（即市场）中等待任务。等到任务到达时，工作线程加入任务调度平台并参与任务的执行。

图 18-4 描述了一个应用中的多个任务调度平台，并举例阐明了由 Intel TBB 库产生工作线程一个应用产生主线程的过程。

随着并发控制和独立任务不会独自对应到应用线程的需求的产生，Intel TBB 4.1 版本对于用户管理（或显式的）任务调度平台引进了一个团体预览特性。用户管理任务调度平台的接口由类 `tbb::task_arena` 提供。当产生一个 `task_arena` 时，用户应具体指明期望的并行程度和需要为应用线程保留多少并行性。

直到 Intel TBB 4.3 版本，作为 Intel TBB 库的预览特性，`task_arena` 要求 `TBB_PREVIEW_TASK_ARENA` 宏在编译时定义；对于不超过 4.2.1 的版本，还需要同预览的二进制相连接。然而，由于 Intel TBB 4.2.1 版本所有必要的功能都存在于常规二进制库中并且

没有要求连接到特别的库。

用户管理任务调度平台示例声明如图 18-5 所示。在这个例子中，任务调度平台由 4 个线程插槽产生，为一个主线程预留一个插槽。

提交任务到 `task_arena` 有两种方法：

- `task_arena::enqueue()`——异步方法，发出后自寻方式，通过提交未加入该调度平台的线程来执行，然后立即返回。

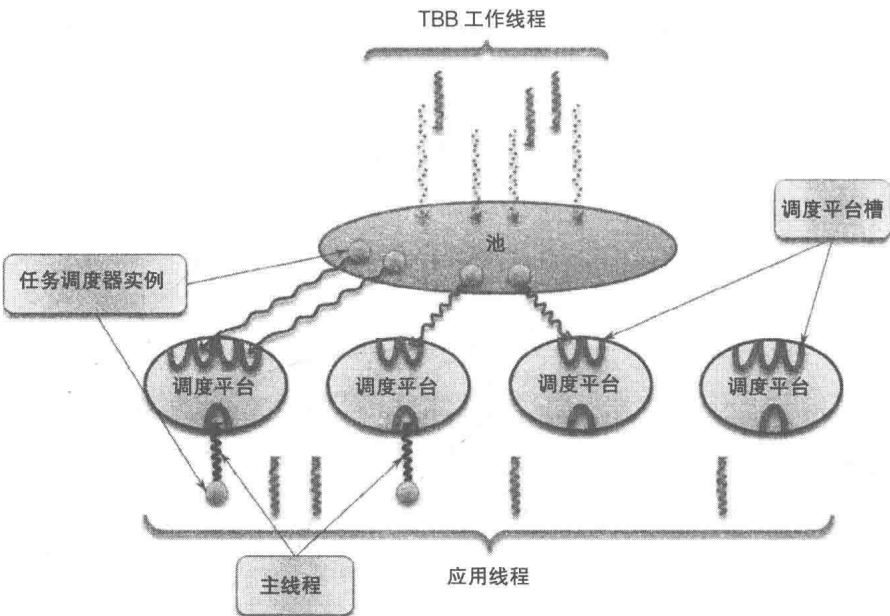


图 18-4 应用内的任务调度平台

```
#define TBB_PREVIEW_TASK_ARENA 1 // for TBB < 4.3
#include <tbb/task_arena.h>
tbb::task_arena user_arena(4,1);
...
```

图 18-5 用户管理任务调度平台的声明

- `task_arena::execute()`——同步方法，它不返回，直到提交任务完成。如果可能，调用者线程加入该调度平台并在那里参与任务的执行，如果线程不能加入该调度平台，在当时被阻塞以完成该调度平台的任务。

一个功能类或者一个 C++11 `lambda` 表达式可以作为工作描述符来论证上面的函数。

图 18-6 为 `task_arena` 提供了一个使用示例。

尽管提交异步作业到 `task_arena` 是可能的，但是并没有显式的方法来确定异步任务完成的时间。这项功能通过使用 `tbb::task_group` 接口来获得。图 18-7 演示了如何利用 `tbb::task_group` 接口来使任务等待执行。

```
user_arena.execute([&] ()
{
    // Some parallel work
    tbb::parallel_for(0,N,[&](int i) {
        // Parallel kernel code
    });
});
```

图 18-6 在调度平台内执行并行循环


```
tbb::task_group waiting_group;

// Submit 1st asynchronous task
user_arena.enqueue([&]{
    waiting_group.run([&]{
        // Some code section #1
    });
});
// Submit 2nd asynchronous task
user_arena.enqueue([&]{
    waiting_group.run([&]{
        // Some code section #2
    });
});
// Do some other work
...
// Join calling thread to arena and wait for completion
user_arena.execute([&]{
    waiting_group.wait();
});
```

图 18-7 异步任务并且等待 task_arena 完成

18.6 分层方法——Hierarchical_arena 类

用户管理任务调度平台接口有一个很好的特点，它提供设置调度平台并发性的功能，在任务被提交到一个调度平台之后，通过控制同时运行任务的数量来完成，只有当获得一个空闲的工作线程时任务才能被分配，并加入到调度平台中。这个性质使用在一个 hierarchical_arena 类里的分层方法中。这个类分配一个最外面的 task_arena 来控制最外面任务的并发性。任务被重新安排到层级中的第二级 task_arena 中，在那里它以数据并行的方式运行。第二级调度平台的并发性是有限的，所以两个层级的并发性乘积与可获取的线程数是相匹配的。

Intel TBB 的默认行为并没有暗示任何线程管理，例如：线程相关性。为了提高数据局部性，通过使进入第二级调度平台的线程会驻留在相同或物理上最近的内核的方式，hierarchical_arena 类实现在最近的逻辑内核上关联工作线程。通过这种方法，数组中最近的索引会被共享相同缓存的硬件线程处理（这会降低 QPI 延迟）和其他有限的内核资源（如 TLB，这也会降低 DRAM 内存访问延迟）。完整 hierarchical_arena 类的实现（包括相关性和其他的优化方法）在 arenas.h 中可获得，参考 18.10 节。

图 18-8 描绘了处理器线程和调度平台间的分配。这个图展示了调度平台的分层结构，第一任务级上有 4 路的并发性，第二级上有 12 路的并发性，总共构成 48 路的并发性。

同基于流水线的实现相反，分层调度平台方法支持动态任务提交，不需要额外的开销并且同所有使用在第一级上的 TBB 算法兼容。不过，由于一个处理器被分割成多个相同的组，因此划分的数量受限于处理器内核的数量除以期望的组大小。

18.7 性能评估

图 18-9 描述了基于流水线和分层调度平台方法修改后的基准测试的性能评估。曲线图表现了线程库的效率，当运行任务总数为 120 时，作为一个同时执行任务的函数，在 Intel Xeon Phi 7120A 协处理器上测量。对于分层调度平台方法，当同时运行的任务数为 20 时获得最优性能；对于流水线方法，当任务数为 25 时达到最优性能。

这些结果同之前图 18-2 所示的线程库分析相匹配，在之前的性能结果中当同时运行的任务数为 20 时达到最高效率，由于存储限制，当同时运行的任务数达到 29 时效率会下降。

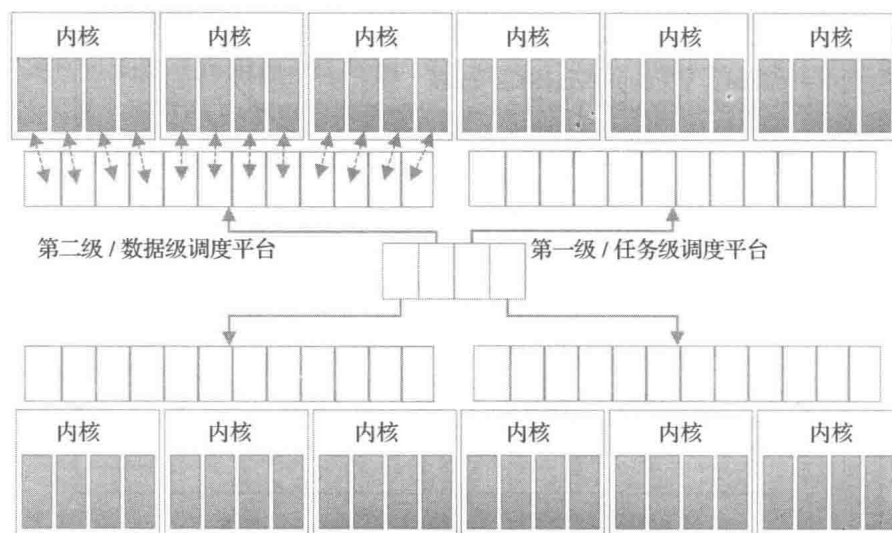


图 18-8 两级分层调度平台（第一级有 4 个槽，第二级有 12 个槽，总共 48 个槽）

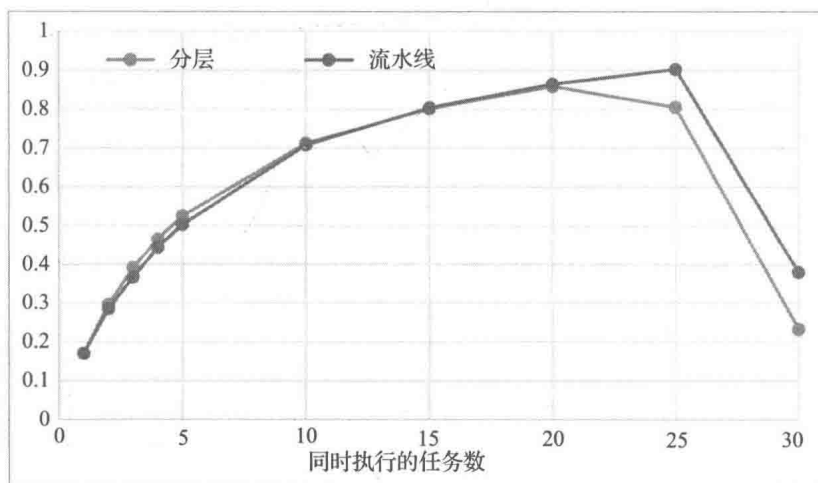


图 18-9 线程模型效率与模拟任务数的关系

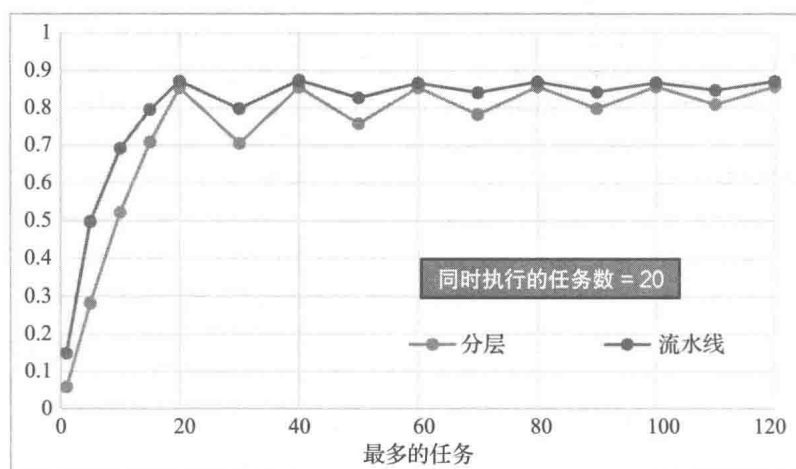


图 18-10 线程模型效率与总任务数的关系

为了更进一步评估最多任务数量的执行效率，基准测试将会配置成针对 20 个任务，或者在 Intel Xeon Phi 7120A 协处理器上，每个任务将利用 3 个内核或者 12 个硬件线程。图 18-10 展示了流水线和分层调度平台方法的效率与最多执行的任务数的关系。120 个任务数的最大值是估值，但是这个值并非是有限的。

在对一些同时运行的任务进行的评估中，在任务数达到 20 时取得最高效率。对于更多数量的任务，流水线方法相较于分层调度平台方法拥有更稳定的结果。这是因为在第一种情况下子任务在一个单独的 task_arena 中分配。而在第二种情况下，子任务的执行仅限于第二级调度平台中。

18.8 对 NUMA 架构的影响

如前几节所提出的，分层调度平台方法通过采用限制同时运行的任务数量，帮助解决了 Intel Xeon Phi 协处理器上的存储限制。此外，这个方法提供了通过线程相关性控制的一种能力，可以在一组预定义的内核上局部化执行一个任务。如果组的数量和 NUMA 节点的数量相等，并且第二级调度平台工作线程已关联到相同的 NUMA 节点，这种方法可以帮助消除不必要的 Intel QPI 传输引起的延迟，并且此外，提供确定的任务执行时间，这对近实时程序是至关重要的。

图 18-11 展示了基于双插槽 Intel Xeon E5-2697v2 处理器系统上的执行结果。“总时间”代表基准测试的实际执行时间，“最短执行时间”代表一个任务的最短执行时间，“最长执行时间”代表一个任务的最长执行时间。

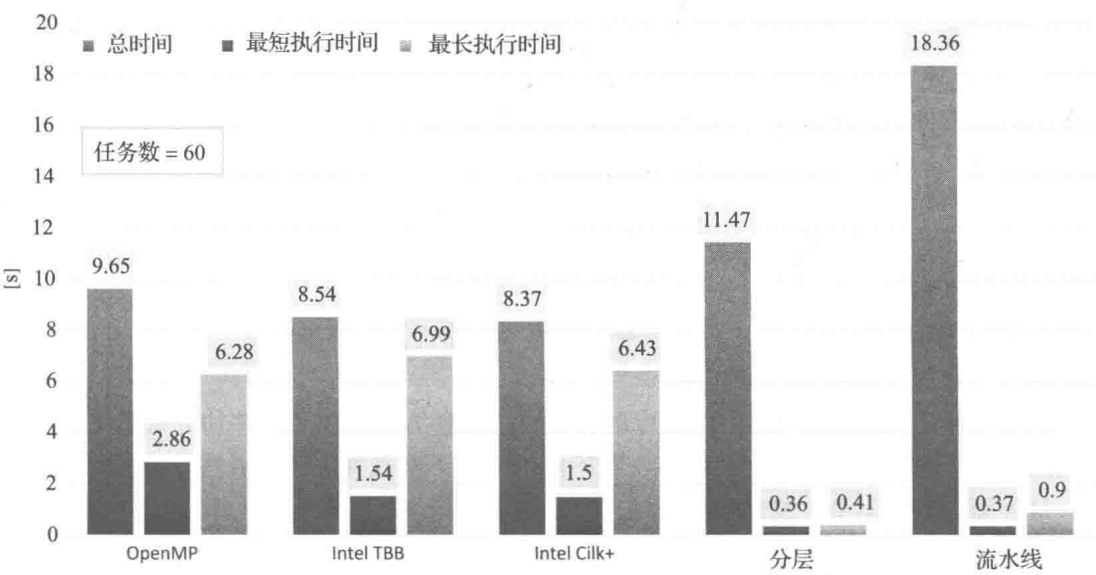


图 18-11 基准测试在双插槽 Intel Xeon E5-2697 v2 上执行

虽然分层方法的实际执行时间比线程库、OpenMP、Intel TBB 和 Intel Cilk+ 的基线低了 30%，但是分层方法显著缩短了时间并能提供确定的任务执行时间。通过流水线（平面调度平台）的方法执行使得任务执行时间相对不稳定，并且实际的执行时间会显著增高。因此后一个方案并不是一个可行的解决方案。

为了理解存储子系统的含义，Intel VTune Amplifier XE 用于粗略估计在基准测试执行

期间，节点间访问发生的数量。获取下面的事件：MEM_LOAD_UOPS_MISS_RETIRED.REMOTE_DRAM——通过 Intel QPI 统计与节点间访问相关联的事件，计数器更高的值表明更高的槽间交流率，于是导致增加了访存延迟；MEM_LOAD_UOPS_MISS_RETIRED.LOCAL_DRAM——统计与本地 DRAM 中的负载相关的事件；MEM_LOAD_UOPS_RETIRED.LLC_HIT——统计 LLC 处理器中的加载，这不需要 DRAM 访问。图 18-12 展示了在基准测试执行过程中测量的计数器值。大多数事件发生在主要计算函数——compute_sqrt_block() 中。

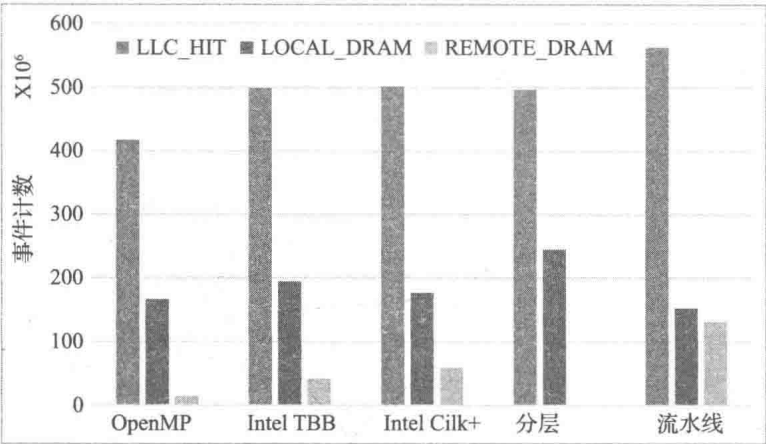


图 18-12 通过基准测试测量的存储子系统计数器（加载）

正如预期，通过分层调度平台方法，避免了远程 DRAM 访问。Intel VTune Amplifier XE 是一个统计工具。因此，在测量 OpenMP 基线时，测量结果依赖于基准测试执行过程中发生的其他事件，这个过程中事件数量的比重相对较低。

18.9 总结

本章探讨了嵌套并行的好处。由于 Intel Xeon Phi 协处理器和 Intel Xeon 处理器所共享的编程方法是高度兼容的，因此这里的方法对于处理器和协处理器均有价值。

我们给出了一个案例研究——一种伪造的“存储饥饿”基准测试，这受 Intel Xeon Phi 协处理器上物理存储器大小的限制。

我们展示了两种基于 Intel TBB 库的方法，这两种方法减弱了一些协处理器上的物理存储器的限制。每种方法有不同的特点，在不同的情况下会选择不同的方法。这些属性需要考虑：最外面的算法类型、机械拓扑结构，以及需要执行的任务数。

最后，我们展示了相同的分层调度平台方法可适用于基于多插槽 Intel Xeon 处理器系统，并能帮助获得确定的执行时间和最小化插槽间通信的延迟。

18.10 更多信息

这里有一些推荐的相关阅读材料：

- STAC A2 基准测试说明和 Intel 审计结果：

<https://stacresearch.com/a2>

<https://stacresearch.com/news/2013/06/23/stac-reports-stac-a2-intel-xeon-and-intel-xeon-phi>

- Kukanov, A., Polin, V., Voss, M.J., 2014. Flow Graphs, Speculative Locks and Task Arenas in Intel® Threading Building Blocks, The Parallel Universe, Issue 18.
https://software.intel.com/sites/default/files/managed/6a/78/parallel_mag_issue18.pdf.
- Voss, M., Wilmarth, T., 2009. Intel® threading building blocks: ready for nonuniform memory access platforms. Intel Softw. Insight Mag. (20).
- 本章或其他章中的代码下载地址 <http://lotsofcores.com>。
- Intel 线程构建模块 TBB, <https://www.threadingbuildingblocks.org/>.
- OpenMP, <http://openmp.org/wp/>.
- Intel Cilk Plus, <http://www.cilkplus.org>.

Black-Scholes 定价的性能优化

IosifMeyerov*, Alexander Sysoyev*, Nikita Astafiev†, IlyaBurylov†

* 俄罗斯, 罗巴切夫斯基州立大学; † 俄罗斯, Intel 公司

虽然高性能计算是计算机科学里一个朝气蓬勃的领域, 但作为一个行业, 它对新优化技术的投资趋于保守——即使这些技术有极大的加速潜力。Intel Xeon Phi 协处理器系列的出现将众核与传统的 x86 处理器体系结构结合在一起, 提供了大规模并行架构的速度, 但需要努力转变应用程序以充分利用硬件所提供的并行性。转变程序的需求也带来了一些问题, 比如什么数据结构和算法才适用? 如何才能最好地扩展应用程序? 什么优化技术带来收益? 什么应用适合这种新的高度并行架构? 能否同时对主机 Intel Xeon 处理器和 Intel Xeon Phi 协处理器的程序进行优化?

本章将介绍利用 Intel Xeon 处理器及 Intel Xeon Phi 协处理器对一套欧式期权进行公平价格计算的优化经验。

本章选择这一主题原因如下: 第一, 欧式期权定价 (European option pricing) 传统上就被用做验证新体系结构能力的基准程序。第二, 它是金融市场分析的基本要素之一, 需要 HPC 系统的计算能力, 因此有很大的实际利益。第三, 实现方法很容易理解。期权定价算法基于流行的 Black-Scholes 公式, 该公式在金融数学教科书中有很好的描述, 不需要任何特别的实现知识。最后, 该算法只有几行代码, 很简单, 但仍旧有实验空间!

仅看 Black-Scholes 算法的描述, 很难想象在实现中仍有易犯的错误和难理解的问题, 这些给优化技术的论证提供了很好的机会。

在这个案例研究中, 本章将逐步讨论各种应用优化方法, 它们对主机处理器及协处理器上其他软件的开发有用。本章将报告所取得的进展及暂时的失败, 包括很有希望但没有取得明显效果的优化方法。将这些优化步骤包含在讨论中的目的是强调, 在其他应用中实现相似的方法能够取得成功的改进。真的很难想象仅仅两百行代码能提供如此多的学习机会!

本章的组织如下。首先将给出简短的金融市场模型描述, 讨论基本概念以确保对算法关键要素的理解。然后描述基本实现, 完成性能分析, 再逐步采用各种优化, 包括: 消除不必要的类型转换, 循环不变代码外提, 将“重量级”数学函数等价替换为“轻量级”、计算向量化, 并行化, “热身”线程创建以避免一次性开销造成结果失真, 降低浮点计算精度, 内存优化 (通过流存储的使用), 最后将展示处理器及协处理器的优化效果。换句话说, 本章首先讨论对所有并行程序最通用的优化, 通过展示处理器运行时间阐明优化效果。直到本章结尾部分才区分协处理器和处理器的区别。这样本章能阐明富有成效的优化方法学: 首先提取通用并行性, 而后专注于特定体系结构的微调。

19.1 金融市场模型基础及 Black-Scholes 公式

19.1.1 金融市场数学模型

考虑一个在连续时间内不断发展的金融市场, 它包括两种类型资产——股票 (基于风险

资产, S) 和债券 (无风险资产, B)。本章采用广泛使用的 Black-Scholes 模型, 该模型经过若干变换, 可以由如下所示的随机微分方程组来表示:

$$dB_t = rB_t dt, \quad B_0 > 0 \quad (19-1)$$

$$dS_t = S_t((r - \delta)dt + \sigma dW_t), \quad S_0 > 0 \quad (19-2)$$

式 (19-1) 是一个普通的微分方程式, 它描述了受 r (利率) 影响的债券 B_t 的价格行为。式 (19-2) 是一个随机微分方程式, 它描述股票 S_t 价格的发展变化。除了利率 r 之外, 该式还包括股息率 δ 、波动率 σ , 以及维纳随机过程 (Wiener Stochastic Process, WSP) $W = (W_t)_{t \geq 0}$ 。初始股票和债券价格 (分别是 S_0 和 B_0) 是预定义常量。

下面简单解释该模型的经济逻辑。第一个式子显示了无风险资产——债券的投资能力。读者可以想象你来到银行存款, 存款利率由银行根据通货膨胀、独特的局势及市场来决定。第二个式子显示两组因素对股票 (风险资产) 价格的影响——确定的及随机的。第一部分—— $S_t(r - \delta)dt$ ——与式 (19-1) 的右侧部分类似, 唯一的不同是, 不像债券, 股息可以用来支付股票, 在等式中由股息率 δ 来反映。

$S_t\sigma dW_t$ 是最有意思的部分, 它以加法的形式包括在式子中, 用于建模市场中随机的、难预测的因素影响。

波动率 σ 表示市场的随机过程: $\sigma=0$ 意味着一切都是确定的, 即没有风险。 σ 越大, 风险越高 (既可能是收益也可能是损失)。乘数 dW_t ——WSP 微分——描述在给定时间点 t 受随机因素影响股票价格的变化。

WSP 是连续时间布朗运动的数学模型, 有如下定义:

1. $W_0=0$ 时概率为 1。
2. W_t ——独立增长过程。
3. $W_t - W_s \sim N(0, t - s)$, 其中 $s < t$, $N(0, t - s)$ 是均值为 0、方差为 $t - s$ 的高斯分布。
4. 过程轨迹 $W_t(\omega)$ ——概率为 1 的时间连续函数。

在以上式子的实现中, 本章有几个关键假设。

- 在接下来的公式和计算中, 为了简化, 假设 $\delta=0$ (即模型不包括股息率)。
- 若时间以年为单位, 则所有式子中的利率都为 0.0 ~ 1.0 间的数字——表达为小数的年利率。
- 假设利率和波动率为常数——它们不依赖于时间。

假定常数利率和波动率, 微分方程式 (19-1) 和 (19-2) 的系统就有解析解, 否则, 需要使用已知的方法之一 (欧拉、龙格-库塔等) 来求数值解。当构造差分格式时, 与 ODE 的唯一不同是, 每个 WSP 增长由从 $N(0, t-s)$ 获取的随机数来建模。式 (19-2) 的解如下:

$$S_t = S_0 e^{\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t} \quad (19-3)$$

19.1.2 欧式期权和公平价格概念

期权是一个衍生的金融工具或者说是 P_1 和 P_2 双方的合同, 它赋予 P_2 在未来的某个时间点 t 以合同中规定的价格 K 向 P_1 购买或出售股票的权利。作为该权利的回报, P_2 将给 P_1 支付一定的数额 (费用) C 。 K 叫作执行价格 (strike price), C 叫作期权价格 (option price)。

本节考虑期权的最简单变型——欧式股票看涨期权 (European share call option)。这个

合同的主旨思想是 P_1 和 P_2 双方的比赛。乙方付数额 C ，并在某一时间点 T (到期，在合同中确定) 做决定：是否以价格 K 从甲方买股票。决策的制定要依据价格 S_T 和 K 的比值。如果 $S_T < K$ ，买股票将无利可图，甲方收益 C ，乙方损失 C 。如果 $S_T > K$ ，乙方以价格 K 从甲方买股票，在一些情况下获得收益 (取决于 C 和 $S_T - K$ 之间的比率)。

主要问题是这种期权合同的公平价格 (fair price) 计算，它在甲乙双方双方的收入和损失达到均衡时发生。将该价格定义为乙方 P_2 的平均收益是合乎逻辑的：

$$C = E(e^{-rT} \cdot \max(S_T - K, 0)) \quad (19-4)$$

在式 (19-4) 中，看涨期权 C 的公平价格由现金流函数 $\max(S_T - K, 0)$ 乘以一个折扣系数 e^{-rT} 的数学期望来获得， e^{-rT} 是在时间段 $(0, T)$ 内利率为 r 时的通货膨胀。

19.1.3 Black-Scholes 公式

在前面的假设下，式 (19-4) 有解析解，即称为欧式看涨期权价格的 Black-Scholes 公式：

$$C = S_0 F(d_1) - K e^{-rT} F(d_2)$$

$$d_1 = \ln \frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right) T}{\sigma \sqrt{T}} \quad (19-5)$$

$$d_2 = \ln \frac{S_0}{K} + \frac{\left(r - \frac{\sigma^2}{2}\right) T}{\sigma \sqrt{T}}$$

其中， F 是累积正态分布函数。

本章将在接下来的计算中使用此公式。读者可能会问：“这有什么可算的？”因为公式中的计算乍一看都很基础，但实际上如下所示，所有计算都不简单。

19.1.4 期权定价

若只给一个期权定价，肯定不需要高性能计算机。但实际上，金融市场中的机构要为数量庞大的不同期权来计算价格，这些期权又是在特定市场条件下发行的。金融计算的时间严重影响决策制定的速度，分秒必争，因此降低一套期权的估价时间是非常重要的因素。

下面构建一个期权的数据依赖图 (见图 19-1)。

构建一套期权一般可能需要改变 5 个参数 (股票初始价格、执行价格、利率、波动率、期限)。但事实上，市场参数 (利率和波动率) 在特定时间点对所有的期权都是相同的，因此，在接下来的计算，本章假设不同期权仅在股票初始价、执行价和期限上有所不同。

19.1.5 测试平台架构

计算实验在下述测试架构上执行 (见图 19-2)。

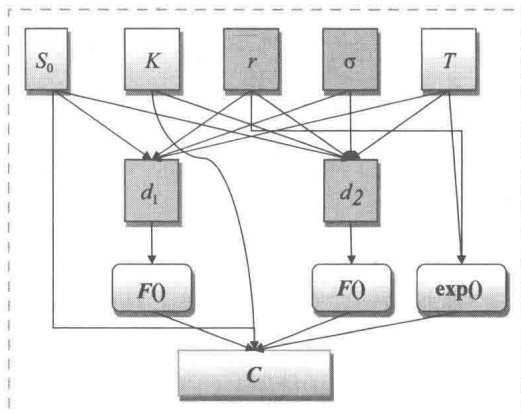


图 19-1 数据依赖图

CPU	两个Intel Xeon E5-2690 处理器 (8核, 2.9 GHz)
协处理器	Intel Xeon Phi 7110X协处理器 (61核)
RAM	64 GB
操作系统	Linux CentOS 6.2
软件	Intel Parallel Studio XE 2013 SP1

图 19-2 测试平台架构

19.2 案例研究

19.2.1 初始版本——检验正确性

参数设置如下 (见图 19-3)。

```
const float sig = 0.2f; // Volatility (0.2 -> 20%)
const float r = 0.05f; // Interest rate (0.05 -> 5%)
const float T = 3.0f; // Maturity (3 -> 3 years)
const float S0 = 100.0f; // Initial stock price
const float K = 100.0f; // Strike price
```

图 19-3 常量

本章一开始用函数 `GetOptionPrice()` 来实现该软件, 该函数使用 Black-Scholes 公式 (即式 (19-5)) 进行期权定价。值得注意的是, 本函数及本章所有其他函数都使用单精度浮点, 这类问题本身不需要使用双精度浮点 (例如, 输入数据通常是单精度的, 类型为 “float”)。接下来读者将看到精度可进一步降低 (见图 19-23)。

式 (19-5) 中较为复杂的部分是累积正态分布函数 F , 这里使用 `cdfnormf()` 函数对其进行估算。

接下来编译执行代码 (见图 19-4) 看其是否工作。函数执行结果一定是 20.924。

```
float GetOptionPrice()
{
    float C;
    float d1, d2, p1, p2;

    d1 = (logf(S0 / K) + (r + sig * sig * 0.5f) * T) /
        (sig * sqrtf(T));
    d2 = (logf(S0 / K) + (r - sig * sig * 0.5f) * T) /
        (sig * sqrtf(T));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    C = S0 * p1 - K * expf((-1.0f) * r * T) * p2;

    return C;
}
```

图 19-4 初始版本

19.2.2 参照版本——选择合适的数据结构

确定应用程序功能正确后, 下面要进入主要目标——估价期权。下面介绍变量 (见图 19-5)。


```
int numThreads = 1;           // The number of threads
                                // (will be useful later)
int N;                        // The number of options
```

图 19-5 变量

首先讨论本程序的数据存储需求，这个似乎比较简单。程序需要 4 个数组（3 个存储输入数据，1 个存储结果），及几个标量变量。

数据在存储中的布局会显著影响软件性能。因此，在和本程序类似的许多应用程序中，一个两难的问题是：该使用 SoA 模式（数组结构）还是 AoS 模式（结构数组）。

- 对于 SoA，数据按下面的方式存放在存储器中：第一个数组作为整个一块来存储，然后是第二个，依次类推。
- 对于 AoS，所有与第一个专业领域对象相关的数据首先保存，然后是所有与第二个专业领域对象相关的数据，依次类推。

通常来说，哪种数据布局更好没有预先确定的答案，但在特定情况下可以有些建议。AoS 模式考虑了内存访问的局部性，能最大限度地重用处理器缓存，但代价是访问单个结构成员时复杂的寻址。使用 SoA 数据布局，缓存可被一些较小任务中的几个数组同时有效使用，这些任务有独立迭代及简单的访存需求。这种布局方式简化了寻址并降低了机器指令总数。本章的代码属于第二种应用类型，使用 SoA 模式可取得显著的性能提升（约 3 倍），如接下来第二个版本的 GetOptionPrices() 函数所示。因此，本章选用 SoA 数据模式。

下面的例子显示了使用 SoA 数据布局的 GetOptionPrices() 函数（见图 19-6）。

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
                    float *pC)
{
    int i;
    float d1, d2, p1, p2;
    for (i = 0; i < N; i++)
    {
        d1 = (log(pS0[i] / pK[i]) + (r + sig * sig * 0.5) *
              pT[i]) / (sig * sqrt(pT[i]));
        d2 = (log(pS0[i] / pK[i]) + (r - sig * sig * 0.5) *
              pT[i]) / (sig * sqrt(pT[i]));
        p1 = cdfnormf(d1);
        p2 = cdfnormf(d2);
        pC[i] = pS0[i] * p1 - pK[i] *
                exp((-1.0) * r * pT[i]) * p2;
    }
}
```

图 19-6 参照版本 1

这里使用 Intel C++ 编译器，用 -O2 选项进行编译，并在 Intel Xeon 主机处理器上运行。执行时间取决于期权数量 N，如图 19-7 所示。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970

图 19-7 参照版本。时间单位为秒

19.2.3 参照版本——不要混合使用数据类型

许多 C 程序员（不仅仅是初学者）会犯的一个典型错误是在处理浮点数时混用“float”

和“double”类型。下面考虑精度如何影响性能，具体考虑当数据表示为“float”类型（32位单精度）且调用数学函数进行计算时需要表示为“double”类型（64位双精度）时的性能影响。注意，图 19-8 的 `GetOptionPrices()` 函数代码调用的是单精度函数 `logf()`。如果程序员调用的是 `log()` 函数，编译器需要执行下面的精度转换：C 语言中的 `log()` 函数传递的参数是“double”类型，返回结果也是“double”类型，因此数组元素 `pT[i]` 首先要从“float”转换为“double”，才能执行所有使用“double”类型的计算。只有在该表达式求值的最后，将结果赋值给 `d1` 时，才会转回“float”类型。考虑到许多情况下“float”类型比“double”类型处理得快（尤其在使用向量化时区别更明显），这些不必要的转换会对总性能产生不利影响。本节会检验两种情况下的性能区别。

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
                    float *pC)
{
    int i;
    float d1, d2, p1, p2;
    for (i = 0; i < N; i++)
    {
        d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) *
              pT[i]) / (sig * sqrtf(pT[i]));
        d2 = (logf(pS0[i] / pK[i]) + (r - sig * sig * 0.5f) *
              pT[i]) / (sig * sqrtf(pT[i]));
        p1 = cdfnormf(d1);
        p2 = cdfnormf(d2);
        pC[i] = pS0[i] * p1 - pK[i] *
            expf((-1.0f) * r * pT[i]) * p2;
    }
}
```

图 19-8 参照版本 2

为此，本节将代码修改为调用函数 `logf()`、`sqrtf()` 和 `expf()`，并恰当地说明常量（意思是如果没有后缀“f”常量 1.0 将会另保存为 double 类型）。代码的更改都用粗体字突出显示了。

使用上面提到的硬件架构，本节得到了图 19-9 所示的执行时间。如图 19-9 所示，与上一个参照版本相比，在所有数据大小的情况下，程序执行时间都微降。请看由 Intel VTune Amplifier XE 收集到的函数 `GetOptionPrices()` 性能分析信息（见图 19-10），以便更好地理解其中的原因。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989

图 19-9 不混用数据类型。时间单位为秒

VTune Basic Hotspots 分析显示，大部分时间都花在了 `cdfnormf()` 函数上，它隐藏了双精度 `exp()`、`log()`、`sqrt()` 计算的开销。然而，这种技术是广泛适用的，本章作者之前也使用过该技术，在特定情况下，程序性能可提高几倍。结论是：如果可能的话，不要混用数据类型！

19.2.4 循环向量化

向量化性能关键的代码是最重要和最有效的优化技术之一。用户可以通过在编译命令

行中加入 `-mavx` 开关令编译器使用 AVX 指令集（`-O2` 开关默认假设只使用 SSE2 指令）。但是，本节的实验证明使用这一选项并没有改变执行时间。

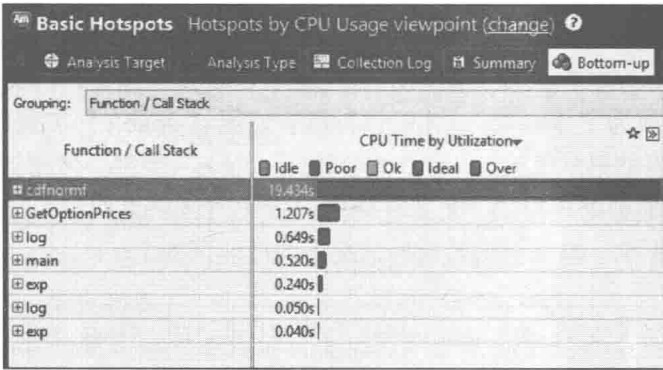


图 19-10 GetOptionPrices() 函数参照版本性能分析结果，使用了 cdfnormf()

出于理解缘由，本节通过 `-vec-report3` 开关为编译器加入向量化报告请求。数字 3 对应报告的详细程度（范围是 1 ~ 6）。注意，第 6 个报告类型（`-vec-report6`）提供关于向量化的一些额外建议。

编译图 19-8 所示代码所产生的向量化报告如图 19-11 所示。编译器报告在代码关键循环中的数组之间可能存在依赖。作者的经验是 Intel C/C++ 编译器给出的诊断经常能帮助程序员精确识别出妨碍向量化的原因。本例还需要进一步的研究以识别出根本问题。

```
sh-4.1$ gcc -O2 -openmp -mavx -vec-report3 main.cpp -o option_prices
main.cpp(179): loop. 3: remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(189): loop. 3: remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(199): loop. 3: remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(209): loop. 3: remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(219): loop. 3: remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
sh-4.1$
```

图 19-11 向量化报告

根据向量化报告，建议编译器进行循环向量化（见图 19-12）。向编译器“解释”数组间不存在依赖的方法有几种。下面对这些方法进行详细介绍。

1. 在声明函数正式参数时使用关键字 `restrict` (`restrict` 是 C99 标准的关键字)。它将通知编译器可以安全地对循环进行向量化，因为没有重叠的访存。

```
for (i = 0; i < N; i++)
{
    ...
}
```

图 19-12 主要计算循环

2. 在循环之前使用 `# pragma ivdep` 指令通知编译器忽略可能的数据依赖。如果编译器能够“证明”存在数据依赖性，该指令会被忽略。要慎重使用 `ivdep`，因为它可能导致那些有很难检测到的“真”依赖的代码在向量化时产生错误的结果。

3. 在循环之前使用 `# pragma simd` 指令。这条指令要求编译器不论是否有依赖都向量化代码。因此，程序员要确保没有任何阻碍向量化的问题。这条指令是最激进的

编译优化，但结果由程序员来承担。只有在确定没有数据依赖存在的情况下才能使用该方法。

4. 除了上述方法之外，也可以指定一个专门的选项 `-ansi-alias`，告诉编译器该程序中没有重叠的（内存中）数组。然而，该选项会影响整个源文件，如果不慎使用，可能会意外“中断”代码。

使用上述任何技术都能成功向量化图 19-8 中的代码。结果见图 19-13。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141

图 19-13 向量化循环。时间单位为秒

读者应该注意到了在一台 Intel Xeon 处理器上，向量化版本的执行时间比标量版本约少 8%。本实验所使用的 Intel Xeon E5-2690 处理器支持 AVX 指令，可以同时处理 8 个单精度浮点元素。那么为什么代码没有加速 8 倍？按推断，在本例中，数据打包到向量寄存器及拆包结果的开销应该可以忽略的，因为使用的 SoA 模式可以连续地将数据保存在存储器中。

为了理解为什么实际的性能比预期的低，本节使用 Intel VTune Amplifier XE 对向量化前后的应用程序进行了性能分析（见图 19-10 和图 19-14）。

查看性能分析信息，注意，`logf()` 和 `expf()` 函数被 Intel 编译器运行时库中 SVML (short vector math library，短向量数学库) 的相应向量化函数所取代了。VTune 也记录了耗时的 `cdfnormf()` 函数仍保持标量计算。这说明了缺少向量化显著加速，因为 `cdfnormf()` 函数占总运行时间的 90%。注意，未来的编译器版本可能会改善 `cdfnormf()` 函数的向量化水平。

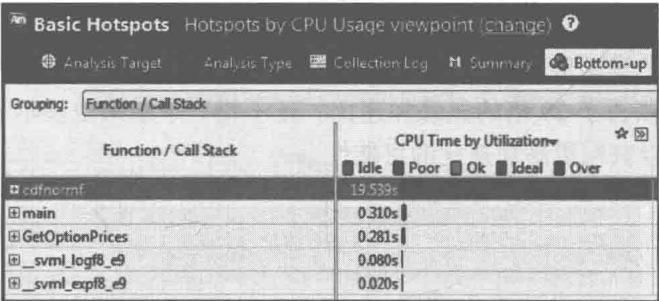


图 19-14 向量化后，`GetOptionPrices()` 函数性能分析结果，使用了 `cdfnormf()`

缺少类似的向量化数学函数是很能说明问题的。不能将热点全部向量化就不能有进一步的提升，尤其是在每个向量能处理 16 个单精度浮点元素的协处理器上。然而，可以通过简单的算术和性能分析器来解决这个问题。

19.2.5 使用快速数学函数：`erff()` 与 `cdfnormf()`

编译器中的一些数学函数通常可能比其他函数有更好的性能优化。在本例中，使用 `cdfnormf()` 函数执行的计算可以使用 `erff()` 函数完成（见图 19-15）。


```
void GetOptionPrices(float *pT, float *pK, float *pS0,
    float *pC)
{
    int i;
    float d1, d2, erf1, erf2;
    for (i = 0; i < N; i++)
    {
        d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) *
            pT[i]) / (sig * sqrtf(pT[i]));
        d2 = (logf(pS0[i] / pK[i]) + (r - sig * sig * 0.5f) *
            pT[i]) / (sig * sqrtf(pT[i]));
        erf1 = 0.5f + 0.5f * erff(d1 / sqrtf(2.0f));
        erf2 = 0.5f + 0.5f * erff(d2 / sqrtf(2.0f));
        pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
            pT[i]) * erf2;
    }
}
```

图 19-15 erff 版本

本例使用下面的公式重新编写了 GetOptionPrices()：

$$\text{cdfnorm}(x) = 0.5 + 0.5\text{erf}\left(\frac{x}{\sqrt{2}}\right) \tag{19-6}$$

使用图 19-2 提到的硬件环境，本例中测量的运行时间如图 19-16 所示。可以预见到 erff() 函数比 cdfnormf() 更快，因为 erff() 的数值性支持更简单的浮点数近似。然而，这不是加速的唯一原因。再次查看 VTune 性能分析结果。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17,002	34,004	51,008	67,970
不混用数据类型	16,776	33,549	50,337	66,989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091

图 19-16 使用快速数学函数。时间单位为秒

图 19-17 显示了编译器采用了 erff() 的 SVML 向量类似函数，因此与 cdfnormf() 的标量代码相比，获得了 29 倍的显著加速比。还不错！本结果也显示了使用支持更宽向量指令的协处理器，以获取更高加速比的可能性。

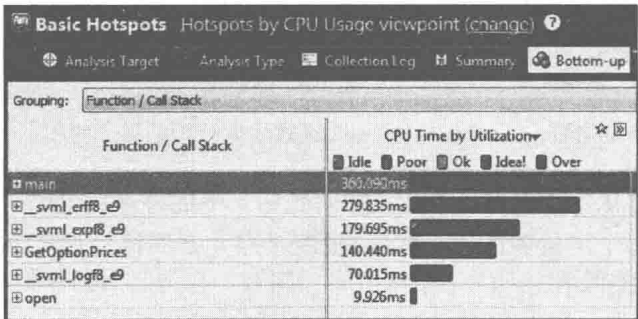


图 19-17 向量化后，GetOptionPrices() 函数性能分析结果，使用了 erff()

19.2.6 代码等价变换

本程序可以通过将“不变的”一次性计算提升到循环外面来做进一步的小优化。在本例

里是 $1.0f/\text{sqrtf}(2.0f)$ 。

可以通过引入下述常量（见图 19-18）单独对该表达式进行估值。

```
const float invsqrt2 = 0.707106781f;
```

图 19-18 “ $1.0f/\text{sqrtf}(2)$ ” 常量

将除法用乘法来替换也是个重要的方法。这种优化在一些情况下可以显著缩短执行时间。本例可以使用 `invsqrtf()` 函数替换 `1/sqrtf()` 表达式。可以检查编译器是否能执行自动替换。

如图 19-19 所示，新代码版本的执行时间几乎与之前得到的执行时间相同。因此编译器非常有可能在无程序员帮助的情况下已经执行了必要的代码替换。可以通过检查编译器的汇编指令来证明这一点（例如在命令行中添加 `-Fa` 开关）。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091
等价变换	0.538	1.071	1.614	2.133

图 19-19 代码等价变换。时间单位为秒

尽管执行时间基本没变，但是本例采用的循环不变式外提及除法的乘法替换一般都很有用，而且对不那么“智能”的编译器，这些方法一般都能带来收益。当代码移植到不同体系结构时，保留这些优化可能会有所帮助，因此本例采用这一版本（见图 19-20）作为新的基准版本，另外在下面的协处理器实验中也将会包含这些优化。

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
                    float *pC)
{
    int i;
    float d1, d2, erf1, erf2, invf;
    float sig2 = sig * sig;

    #pragma simd
    for (i = 0; i < N; i++)
    {
        invf = invsqrtf(sig2 * pT[i]);
        d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
              pT[i]) * invf;
        d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
              pT[i]) * invf;
        erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
        erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
        pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
              pT[i]) * erf2;
    }
}
```

图 19-20 新的基准版本

19.2.7 数组对齐

另一个可能有用的软件优化方法涉及数据对齐。处理器执行按 SIMD 寄存器大小对齐的

数据访问要比执行非对齐的访问快得多。一些情况下，编译器和硬件能最小化性能影响，但确保数据对齐通常能取得显著的性能提高——尤其对于向量代码，因此值得去检测内存地址是否对齐。为确保对齐，本节将 new/delete 操作符替换为 memalign()/free() 函数，代码的其余部分没有改变（见图 19-21）。

```
int main(int argc, char *argv[])
{
    pT = (float *)memalign(32, 4 * N * sizeof(float));
    // pT = new float[4 * N];

    ...

    free(pT);
    // delete [] pT;
    return 0;
}
```

图 19-21 数据对齐

建议的对齐值（memalign() 函数的第一个参数）取决于所使用 SIMD 寄存器的宽度。对 SSE 指令，使用 16 字节；对 AVX 指令，使用 32 字节；对于协处理器指令集，使用 64 字节。在代码中循环之前加入一条指令也是很有用的：

```
#pragma vector aligned
```

这条指令会通知编译器循环中的数组是对齐的，可以使用相对应的内存读 / 写指令。要注意的是，如果程序员“欺骗”编译器，软件在尝试对非对齐数据使用对齐访问时可能会崩溃。

在本例中，编译器已经处理了对齐问题：图 19-22 显示了最后两组实验没有差别，但这并不意味着对所有编译器会一直有这样的结果。作者的经验表明应该尽可能将数据对齐。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091
等价变换	0.538	1.071	1.614	2.133
对齐数组	0.539	1.072	1.617	2.135

图 19-22 对齐数据。时间单位为秒

19.2.8 尽可能降低精度

之前已经讲到了本问题的解决过程不需要使用 double 类型。本节将进一步讨论这个问题。本例中使用单精度 float 类型也多余了（因为在本专业领域不会使用超过 4 位十进制数字）。降低精度为进一步优化计算提供了可能性。

下面的 Intel 编译器命令行选项会影响数学函数的计算精度。

```
icc ... -fimf-precision=low -fimf-domain-exclusion=31
```

-fimf-precision=low 选项通知编译器使用尾数为 11 个精确位（对于单精度浮点可用 24 位）的数学函数实现，这更与输入参数的精度相符。-fimf-domain-exclusion 选

项允许数学函数不处理一些特殊值（如无穷大值、NaN 及变量的极值），在能安全假定程序不需要处理这些极值的情况下可以使用该选项。

图 19-23 显示了使用这些较低精度的编译器选项可以将应用程序运行时间降低 23%。注意，精度的降低会影响 GetOptionPrices() 函数的数字结果：这里得到的是 20.920，而不是参照版本的 20.924，但直到第 5 个十进制数字才有区别。但要特别指出的是，应该总是执行恰当的精度分析以确保安全地使用精度控制。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091
等价变换	0.538	1.071	1.614	2.133
对齐数组	0.539	1.072	1.617	2.135
降低精度	0.438	0.871	1.314	1.724

图 19-23 尽可能降低精度。时间单位为秒

19.2.9 并行工作

就像对单线程应用来说，向量化能够对可用的处理器核资源饱和有所帮助，线程级并行能够对将工作分给多个处理器内核及 SMT 架构中的可用单核资源有所帮助。计算部分的循环在多个处理单元（例如硬件线程和处理器内核）上的并行化是很简单的，因为迭代之间不存在任何依赖。只要简单地在循环之前加 `omp parallel for` 编译指示（在图 19-24 中由黑体标出）并通过 `private` 列表局部化所有需要写的变量即可。

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
float *pC)
{
    int i;
    float d1, d2, erf1, erf2, invf;
    float sig2 = sig * sig;
    #pragma simd
    #pragma omp parallel for private(d1, d2, erf1, erf2,
    invf)
    for (i = 0; i < N; i++)
    {
        invf = invsqrtf(sig2 * pT[i]);
        d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
        pT[i]) * invf;
        d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
        pT[i]) * invf;
        erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
        erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
        pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
        pT[i]) * erf2;
    }
}
```

图 19-24 OpenMP 版本

注意，`-openmp` 编译器命令行开关之前已经加过了，因此不需要格外的命令行改动。如图 19-25 所示，随着数据量的增加，性能扩展也有所提高，2.4 亿样本的性能扩展为

7.59 ~ 11.27 倍。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091
等价变换	0.538	1.071	1.614	2.133
对齐数组	0.539	1.072	1.617	2.135
降低精度	0.438	0.871	1.314	1.724
并行工作（16核）	0.058	0.084	0.126	0.153

图 19-25 并行工作。时间单位为秒，使用了 16 核

19.2.10 使用热身

上一个实验的执行时间已经很短了，因此线程创建开销可能在并行部分执行时间中占显著比例。下面尝试避免这一开销。大多数 OpenMP 实现并不会销毁为第一段并行代码创建的线程，而是将它们置于睡眠状态，以便在后续使用中可以更快地重新开始。这就是大家所知的创建“线程池”。本方法正是利用这一事实来避免线程创建开销。因此，如果 `GetOptionPrices()` 在 `main()` 函数中连续调用两次，第二次的调用开销将最小化。这种方法叫作“热身”。

热身对基准测试公平吗？作者的经验是实际的产品程序都会在公式计算执行之前很长一段时间就创建线程，因此避免线程创建开销对于基准测试是合理的。然而，数据缓存热身可以节省更多的开销：使用相同数据的相同函数的连续两次调用可能导致不同的运行时间，因为第一次函数调用可能受缓存缺失影响，而第二次就可能受益于缓存命中，因为数据仍保存在高速缓存中。

这样是好还是坏？一方面，这样做不完全公平：有一个非常知名的基准测试错误——在单一进程中执行单线程和多线程实验（操作相同数据集）。当实验正确执行时——单线程和多线程使用自己的进程，超线性加速比立即就消失了。另一方面，在实际的程序中，其他配置计算通常在热点函数周围执行，逐步将目标计算数据载入缓存中。

一般的建议如下：

1. 如果执行时间长，如数秒，就不需要热身。
2. 如果执行时间短（几分之一秒），线程数大，使用热身较合理，否则现实世界中算法的执行时间会被开销掩盖。
3. 如果怀疑预加载缓存可能造成性能结果扭曲，程序员可以只出于线程创建目的，在被测试代码部分之前引入一个专门的假并行部分。这种方法仅避免了线程创建开销，却没有接触到缓存。

图 19-26 所示的“降低精度”和“降低精度 + 热身缓存”的结果差别显示了“热身”缓存的好处。热身也可以从并行版本的测试结果中避免了线程创建开销（图 19-26 中最后一行），可以看到计算部分在 16 核上性能扩展了 13 倍。

N	60 000 000	120 000 000	180 000 000	240 000 000
参照版本	17.002	34.004	51.008	67.970
不混用数据类型	16.776	33.549	50.337	66.989
向量化循环	15.445	30.977	46.608	62.141
使用快速数学函数 +改进的向量化	0.522	1.049	1.583	2.091
等价变换	0.538	1.071	1.614	2.133
对齐数组	0.539	1.072	1.617	2.135
降低精度	0.438	0.871	1.314	1.724
降低精度+热身缓存	0.409	0.812	1.226	1.603
并行工作 (16核)	0.058	0.084	0.126	0.153
并行、热身缓存、避免线程创建开销	0.033	0.062	0.091	0.118

图 19-26 使用热身。时间单位为秒

19.2.11 使用 Intel Xeon Phi 协处理器实现轻松移植

该协处理器已经充分展示了其使用大量线程（120 ~ 240 个）的有效性，前提是如果代码也能很好地向量化。下面讨论只是简单重新编译的“轻松移植”能带来的性能收益。另外，本节也将分析应用于主机处理器的优化在此处的影响。实验从“等价变换”版本开始，将其作为基准版本。

为了编译协处理器的代码，只需简单地在命令行加入 -mmic 开关。另外，别忘了将 memalign() 中的对齐参数从 32 改为 64，以适应协处理器体系结构。

图 19-27 包括了协处理器串行版本与本机处理器串行版本的时间比较。值得注意的是，在协处理器上降低精度的计算能得到 2.3 倍的收益，而 CPU 只有 23%。该图也显示了热身缓存能带来更好的收益（约 60%）。最后，值得注意的是，协处理器串行版本的执行时间与本机处理器的执行时间基本是相同的。

N	60 000 000	120 000 000	180 000 000	240 000 000
最佳的串行CPU版本	0.409	0.812	1.226	1.603
等价变换	1.544	3.089	4.633	6.174
对齐数组	1.545	3.091	4.634	6.179
降低精度	0.676	1.352	2.027	2.703
降低精度+热身缓存	0.422	0.845	1.269	1.690

图 19-27 使用 Intel Xeon Phi 协处理器。时间单位为秒，串行版本

19.2.12 使用 Intel Xeon Phi 协处理器实现并行工作

当所有并行资源都使用时，协处理器的潜力才能显现出来：包括 SIMD 和具有 SMT 的多核。下面讨论协处理器的并行版本。要注意的是，到目前为止，本章还没有采用任何

专门针对协处理器的优化，只是重新编译了处理器的代码版本并在协处理器上以本机模式 (native mode) 运行。

如图 19-28 所示，没有热身的版本基本没有加速，这一结果支持了线程创建开销与程序执行时间相当的假设。

N	60 000 000	120 000 000	180 000 000	240 000 000
并行工作，60个线程	0.134	0.149	0.164	0.175
加速比	5.0336	9.050	12.331	15.437
60个线程，无开销	0.008	0.017	0.025	0.033
加速比	50.585	51.178	51.783	51.546

N	60 000 000	120 000 000	180 000 000	240 000 000
并行工作，60个线程	0.234	0.255	0.257	0.255
加速比	2.885	5.303	7.883	10.590
120个线程，无开销	0.007	0.014	0.021	0.028
加速比	59.422	59.587	60.389	59.839

N	60 000 000	120 000 000	180 000 000	240 000 000
并行工作，240个线程	0.532	0.527	0.533	0.558
加速比	1.269	2.564	3.800	4.842
240个线程，无开销	0.008	0.016	0.024	0.031
加速比	53.286	54.248	53.969	53.964

图 19-28 使用 Intel Xeon Phi 协处理器。时间单位为秒，分别使用 60、120、240 个线程

增加了热身，在测量中即可避免一次性的开销，并且可以看到计算部分性能如何扩展：加速从 50.5 变化到 60.4。需要注意的是，直到线程数量达到内核数，性能都有很好的扩展，若继续增加线程数量以充分利用每个内核的执行流水线，在 120 个线程时（即每个核两个线程），性能也有些许提升，但在 240 个线程时（即每个内核 4 个线程），性能下降了，如图 19-28 所示。很显然，在主机处理器上对于 16 个线程能取得很好性能的代码不能很好地扩展到 240 个线程。下面将讨论是否能针对协处理器改进原代码版本。

19.2.13 使用 Intel Xeon Phi 协处理器和流存储

GetOptionPrices() 函数使用 4 个数组 (pT, pK, PS0, pC)，其中三个数组作为只读的输入，一个数组 (pC) 是有写操作的输出。请注意，当前的基准程序数据组织只在循环中访问 pC 数组一次，因此，pC 数组数据不需要缓存，并且可以标记为非暂存的 (nontemporal)。换句话说，现在得到的结论是该示例已经达到了存储带宽极限，因此即使线程数增多性能也不能扩展。

协处理器上的流存储指令能通过消除缓存一致性流量（称为“带所有权的读 (read-for-ownership, RFO)”（流量））来节省带宽。计算 X 份期权需要 $3X$ 次读和 X 次写，以及 X 次 RFO 请求以维持写的缓存一致性，总共 $5X$ 次操作。使用流存储能消除 RFO 请求，将流量降低到 $4X$ 。如果基准程序是内存带宽受限的，那么该优化中的期望加速比应为 $5X/4X = 1.25$ 。在最大数据集的情况下，本实验可得加速比 $0.031/0.026=1.19$ ，见图 19-29。流存储版

本的代码见图 19-30。

N	60 000 000	120 000 000	180 000 000	240 000 000
最佳并行CPU版本	0.033	0.062	0.091	0.118
并行工作+热身, 120个线程	0.007	0.014	0.021	0.028
并行工作+热身, 240个线程	0.008	0.016	0.024	0.031
并行工作+热身, 流存储120个线程	0.007	0.013	0.019	0.026
并行工作+热身, 流存储240个线程	0.007	0.013	0.019	0.026

图 19-29 使用 Intel Xeon Phi 协处理器和流存储。时间单位为秒

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
float *pC)
{
    int i;
    float d1, d2, erf1, erf2, invf;
    float sig2 = sig * sig;
    #pragma simd
    #pragma vector nontemporal
    #pragma omp parallel for private(invf, d1, d2, erf1,
    erf2)
    for (i = 0; i < N; i++)
    {
        invf = invsqrtf(sig2 * pT[i]);
        d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
        pT[i]) * invf;
        d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
        pT[i]) * invf;
        erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
        erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
        pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
        pT[i]) * erf2;
    }
}
```

图 19-30 流存储版本

最终，使用协处理器评估 240 000 000 份期权比使用主机处理器大约快 4.5 倍，但由于达到了存储带宽极限，程序就不能扩展到高于 65.4 倍了。

19.3 总结

欧式期权定价的 Black-Scholes 公式是事实上的标准金融基准测试程序。尽管由 Black 和 Scholes 发明的模型中的几个假设在现实生活中很少满足，但是 Black-Scholes 公式在实践中仍广泛使用。要解决的一个典型问题是同时为数百万期权定价。这个问题非常耗时，需要大量的计算能力，因此，它与 HPC 领域相关。这引发了在当代多核和众核硬件上以高性能实现 Black-Scholes 公式的挑战。

本章介绍了 Black-Scholes 公式计算在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上的性能优化研究，从基本实现到高度调整的实现。一步步分析了典型优化技术在处理器和协处

理器上所带来的性能提升。在单精度浮点数的情况下，最终优化版本在处理器上可以每秒计算 20.67 亿份期权，在协处理器上每秒可以计算 92.31 亿份期权。本章也阐述了通过应用优化性能制约因素如何变化——即处理器上的应用是计算制约的，而在计算密集的数学计算被调整之后，协处理器上存储带宽似乎成为了制约因素。

这一研究及源码已在俄罗斯作为“Intel Xeon Phi 协处理器编程”教程的一部分发表 (<http://hpc-education.unn.ru/ru/obuchenie/courses/xeon-phi>)，代码参见：<http://lotsofcores.com>。这一工作由 UNN HPC 中心与 Intel Numerics 工程师合作筹备。

19.4 更多信息

- Knuth, D., 1997. The Art of Computer Programming, Seminumerical Algorithms, vol. 2, third ed. Addison-Wesley Professional, 784 p.
- Li, S. Achieving Superior Performance on Black-Scholes Valuation Computing using Intel® Xeon Phi™ Coprocessors. <https://software.intel.com/en-us/articles/case-study-achieving-superiorperformance-on-black-scholes-valuation-computing-using>.
- Jeffers, J., Reinders, J., 2013. Intel® Xeon Phi™ Coprocessor High Performance Programming. Copyright, Morgan Kaufman.
- Smelyanskiy, M., Sewall, J., Kalamkar, D.D., et al., 2012. Analysis and optimization of financialanalytics benchmark on modern multi- and many-core IA-based architectures. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 1154-1162.
- Bastrakov, S., Meyerov, I., Gergel, V., et al., 2013. High performance computing in biomedical applications. *Procedia Comput. Sci.* 18, 10-19.
- Black, F., Scholes, M.S., 1973. The pricing of options and corporate liabilities. *J. Polit. Econ.* 81(3), 637-654.

使用 Intel COI 库传输数据

Louis Feng

美国, Intel 公司

利用 Intel Xeon Phi 协处理器的计算能力有很多方法。例如卸载模式。这种模式下, 编译器自动生成可在协处理器上运行的代码; 同时数据传输自主管理并且执行过程对程序员不可见。对于多种程序, 特别是遗留应用程序 (legacy application) 中的计算内核以及热点内核中的紧耦合计算, 卸载模式不但可以非常容易地改写代码, 而且能够取得较好性能。同时, 当整个程序高度并行并且可以完全运行在协处理器上时, 可在协处理器上编写、编译单机程序 (例如本机模式)。对于适合主从构架或者其他形式的松散耦合计算的应用程序, 划分计算并运行在最合适的硬件架构上是有意义的。

如果计算可以划分为分别针对主机处理器和协处理器的独立进程, 并且进程间有细粒度通信, Intel 协处理器卸载基础架构 (Intel COI) API 和库是一个处理数据传输的好方法。COI 库构建于对称通信接口之上, 为优化后的底层进程和使用 Intel 众核平台软件栈 (Intel MPSS) 的协处理器提供通信。与汇编辅助卸载机制 (也使用的是 COI 库) 相对比, 程序员直接使用 COI 库手工控制数据, 可以直接在协处理器上写入或读出数据。本章将介绍如何使用 COI 缓冲区来传输数据, 评估 COI 库在实际应用中的实效性, 并通过基准测试讨论不同种类 COI 缓冲区的特点。

20.1 使用 Intel COI 库的第一步

COI 库是为了主机和协处理器上进程间通信所设计的。在 COI 术语中, 某个进程是源 (source), 其他进程是汇 (sink)。它们之间的通信通道是一条从源开始通向汇的管道 (pipeline)。源和汇是两个二进制文件, 可针对它们各自的架构执行编译和构建。源进程负责通过 COI 接口调用, 开启协处理器进程。图 20-1 展示了初始化源进程的代码。

```
uint32_t engineCount = 0;
COIEngineGetCount(COI_ISA_MIC, &engineCount);
if (engineCount < 1) {
    return -1;
}
COIENGINE engine = NULL;
COIEngineGetHandle(COI_ISA_MIC, 0, &engine);
COIPROCESS proc = NULL;
const char* sinkName = "sink_mic";
COIProcessCreateFromFile(engine, sinkName, 0, NULL,
    false, NULL, true, NULL, 0, NULL, &proc);
const char* funcName = "receiveData";
COIFUNCTION func[1];
COIPIPELINE pipeline;
COIPipelineCreate(proc, NULL, 0, &pipeline);
COIProcessGetFunctionHandles(proc, 1, &funcName, func);
```

图 20-1 在源端 (主机) 初始化 COI 代码

主机列举出协处理器 (COIENGINE), 初始化并产生协处理器进程 (COIPROCESS), 然后在 COI 管道中声明对应功能的指针 (COIFUNCTION)。一旦这些步骤完成, 我们就可以开始使用 COI 缓冲区在进程间传输数据。

在协处理器端, 只需要发起两个 COI 接口调用, 就可以开启汇的主要功能, 如图 20-2 中的代码所示。汇进程需要使用 COINATIVELIBEXPORT 宏来导出函数, 源进程可以从主机端通过管道调用“执行函数”(run function)来导出函数。导出的 COI 执行函数参数必须同 COI API 指定的执行函数原型相匹配。

```
int main(int , char**)
{
    COIPipelineStartExecutingRunFunctions();
    COIProcessWaitForShutdown();
    return 0;
}
COINATIVELIBEXPORT
void receiveData(uint32_t bufferCount,
                 void** bufferPointers,
                 uint64_t* bufferLengths,
                 void* miscData,
                 uint16_t miscDataLength,
                 void* returnValue,
                 uint16_t returnValueLength)
{
    // Process data buffers...
}
```

图 20-2 在汇(协处理器)侧 CIO 的初始化代码

注意 COI 库是作为 Intel 众核平台软件栈 (Intel MPSS) 的一部分发布的。安装之后, 可以在 Linux 上 /usr/share/doc/intel-coi-<version> 目录下发现 API 文档和源代码教程。

20.2 COI 缓冲区种类和传输性能

COI 源和汇代码使用管道来通信和传输数据。管道命令是按顺序执行的, 并且可同步执行也可异步执行。可以通过使用 COI 完成事件机制来阻塞执行, 直到一个管道事件发生来完成同步。实际上, 在两种情况下, 数据可以通过管道执行函数调用而输送, 指出这一点很重要。一种是使用缓冲区指针参数; 另一种是使用混合数据参数。混合数据在可以传输的数据总量中是有限的 (默认情况下是 32KB), 但是混合数据是传输小块数据最有效的方式, 如传输命令参数而不需要分配 COI 缓冲区。

在 COI 库中有 4 种缓冲区类型用来数据传输。每种在它们的用法和性能上都略有不同。

- 流式 (从 Intel MPSS 3.2 之后弃用)
- 普通
- 固定
- OpenCL

我们将关注前三种缓冲区。OpenCL 缓冲区同普通的缓冲区相类似, 不同之处是前者使用于 Intel OpenCL SDK。所有的 COI 缓冲区都使用 COIBufferCreate 函数创建。当一个

缓冲区不再需要了，使用 COIBufferDestroy 来清理资源。图 20-3 展示了一个使用 COI 流式缓冲区的例子。数据传输的方向是从源到汇。

```
COIBUFFER buffer;
COIBufferCreate(bytes, COI_BUFFER_STREAMING_TO_SINK,
                0, data, 1, &proc, &buffer));
COIEVENT completion_event;
COI_ACCESS_FLAGS flag = COI_SINK_READ;
COIPipelineRunFunction(pipeline, func[0], 1, &buffer,
                       &flag, 0, NULL, misc_data, strlen, return_value,
                       strlen, &completion_event);
// do something else.
COIEventWait(1, &completion_event, -1, true, NULL, NULL);
COIBufferDestroy(buffer);
```

图 20-3 COI 流式缓冲区的创建和数据传输。data 参数在 COIBufferCreate 函数中是一个指向用户数据的指针，缓冲区从该指针处开始复制数据。缓冲区的尺寸是通过 bytes 参数指定的。通过传入一个 COIEVENT 对象，管道执行函数可以异步调用

如 COI 参考手册所述，一个管道执行函数使用一个流式缓冲区。虽然 COIBufferMap 可以通过映射流式缓冲区来完成写操作，但是会有一个新的缓冲区在高级选项中创建。因此，通过 COIBufferMap 重用一個流式缓冲区是有开销的（见图 20-4）。相比之下，使用他三种缓冲区类型创建一个缓冲区，在数据传输中可以重用，直到手工销毁它。本章中的 COI 缓冲区基准测试在一个系统中收集，该系统拥有两个 Intel Xeon E5-2697 v2 2.7GHz 处理器和一个使用 Intel MPSS 3.2.1 的 Intel XeonPhi 7120A 协处理器。

数据大小 (MB)	缓冲区创建带宽		缓冲区传输		缓冲区映射	
	创建时间	带宽	传输时间	带宽	映射时间	带宽
1	1.03 ms	970 MB/s	0.34 ms	2941 MB/s	0.48 ms	2096 MB/s
4	6.36 ms	629 MB/s	0.90 ms	4469 MB/s	2.98 ms	1340 MB/s
16	22.31 ms	717 MB/s	2.83 ms	5662 MB/s	9.73 ms	1644 MB/s
64	112.77 ms	568 MB/s	10.26 ms	6237 MB/s	40.04 ms	1598 MB/s
256	448.88 ms	570 MB/s	39.78 ms	6436 MB/s	158.78 ms	1612 MB/s
1024	1804.06 ms	568 MB/s	158.50 ms	6461 MB/s	641.45 ms	1596 MB/s

图 20-4 COI 流式缓冲区的性能

当有大量的数据需要传输时，重用一個非流式缓冲区会比为每一次数据传输创建一个新的缓冲区更有效。普通缓冲区是以类似流式缓冲区的方式创建的，但是它们的用法不同。虽然 COIBufferMap 可以用来存取缓冲区数据，但是如果我們只想复制一块新的数据到缓冲区里，我们可以使用 COIBufferWrite，如图 20-5 所示。

普通缓冲区和固定缓冲区在缓冲区分配方式上不同，在数据从源传输到汇的方式上也不同。对于一个普通缓冲区，当缓冲区在源端创建时，内存仅在主机端分配，协处理器端的虚拟内存被保留。所以，在源端写入缓冲区是一个本地操作。COIPipelineRunFunction 第一次调用时，在协处理器端分配一个相应的缓存以存储实际传输的数据。这个缓存将会在后续执行函数调用时重用（见图 20-6）。


```
COIBUFFER buffer;
COIBufferCreate(bytes, COI_BUFFER_NORMAL,
                0, data, 1, &proc, &buffer));
COIBufferWrite(buffer, 0, data, bytes, COI_COPY_USE_DMA,
               0, NULL, NULL);
COI_ACCESS_FLAGS flag = COI_SINK_READ;
COIEVENT completion_event;
COIPipelineRunFunction(pipeline, func[0], 1, &buffer,
                       &flag, 0, NULL, misc_data, strlen, return_value,
                       strlen, &completion_event);
// do something else.
COIEventWait(1, &completion_event, -1, true, NULL, NULL);
COIBufferDestroy(buffer);
```

图 20-5 COI 普通缓冲区创建、写入、传输

数据大小 (MB)	缓冲区创建带宽		缓冲区传输		缓冲区映射	
1	0.74 ms	1353 MB/s	0.20 ms	5128 MB/s	0.41 ms	2427 MB/s
4	2.19 ms	1830 MB/s	1.18 ms	3396 MB/s	0.88 ms	4555 MB/s
16	10.09 ms	1585 MB/s	5.16 ms	3098 MB/s	2.74 ms	5833 MB/s
64	54.11 ms	1183 MB/s	23.78 ms	2691 MB/s	10.24 ms	6249 MB/s
256	217.96 ms	1175 MB/s	95.03 ms	2694 MB/s	39.78 ms	6435 MB/s
1024	872.55 ms	1174 MB/s	377.83 ms	2710 MB/s	158.53 ms	6460 MB/s

所有的数据均为在初始化调用一次后管道运行函数的测试结果。

图 20-6 COI 普通缓冲区性能

如图 20-5 所示，一个固定缓冲区的创建和使用过程同普通缓冲区的实施过程相类似。为了创建一个固定缓冲区，只须替换 COIBufferCreate 函数的 buffer-type（缓冲区种类）参数，从 COI_BUFFER_NORMAL 改为 COI_BUFFER_PINNED。图 20-5 中余下部分的代码可以保持不变。顾名思义，当使用一个固定缓冲区时，对于主机和协处理器，所有的内存都会预先分配。它同普通缓冲区在如何获得传输数据的方式上不同。在管道执行函数调用期间，并非从源到汇复制和缓存整个缓冲区，而是汇可以通过分页机制获取固定缓冲区的数据。也就是说，在页面级别的访问上传输数据。图 20-7 中的结果可以证明，不论传输数据量的大小，缓冲区的传输时间仍然是相对稳定的。

数据大小 (MB)	缓冲区创建带宽		缓冲区传输		缓冲区映射	
1	0.62 ms	1618 MB/s	0.32 ms	3144 MB/s	0.22 ms	N/A
4	2.18 ms	1838 MB/s	1.06 ms	3780 MB/s	0.23 ms	N/A
16	10.40 ms	1539 MB/s	5.05 ms	3169 MB/s	0.25 ms	N/A
64	52.29 ms	1224 MB/s	24.57 ms	2605 MB/s	0.28 ms	N/A
256	231.40 ms	1106 MB/s	98.04 ms	2611 MB/s	0.31 ms	N/A
1024	947.81 ms	1080 MB/s	391.85 ms	2613 MB/s	0.36 ms	N/A

注意，“缓冲区传输带宽”一列并没有对应的数据，这是因为缓冲区中的数据在针对固定缓冲区的 COI 调用期间并没有传输。

图 20-7 COI 固定缓冲区的创建、写入和传输性能

由于固定缓冲区数据在基准测试的汇端没有访问，因此实际上并没有缓冲区数据传输并且没有测量带宽。如果一小部分固定缓冲区数据需要通过汇来访问，并且没有什么好的方法来事先断定这些数据可能是什么，这个特殊模型才可能是有用的。

对于使用固定缓冲区的一个警告是，因为内存地址空间在主机和协处理器上都是固定的，所以全部固定缓冲区的大小局限于协处理器上物理内存的大小。如果不能慎重管理，协处理器内存可能会很快耗尽。

20.3 应用程序

当我们通过把主机处理器和协处理器的不同架构当做不同目标来分离计算时，我们获得了很多好处。

- 易维护性：通常每个进程在系统中都有明确的角色，并且不同的组件需要通过一套 API 来通信，这就强制执行了封装。
- 灵活性：通过对程序去耦合，可以在不同的架构上执行程序的不同部分。
- 性能：一些组件在主机系统的处理器上的性能比较好，另一些则在协处理器上通过高度并行和广泛的 SIMD 架构而获得更多优势。

这种程序的一个示例是开源代码 Intel Embree 渲染器和高性能射线跟踪库（见第 21 章）。渲染计算本质上是一种物理模拟。它试图在虚拟 3D 世界计算光的传播方式。计算基于光在物理世界传播方式的粒子模型。射线跟踪是一个高度并行的算法，由于每个射线计算都是独立于其他射线计算的，并且可以很好地扩展在协处理器的大量线程上。

Embree 射线跟踪内核是一组加速数据结构，并将进一步优化，以充分利用协处理器向量处理能力的优势。渲染器的另一部分是使用 Intel ISPC 开源编译器来写的，以充分利用线程和 SIMD 向量处理的优势。总的来说，Embree 渲染器和内核库在协处理器架构上运行得非常好。渲染器为了可视化的目的提供了一个 GUI（图形用户界面）。这个渲染器也在梦工厂动画公司使用，那里提供了一种交互式光线传播模拟来照亮 3D 场景。相机、灯光、几何物体需要处理并发送到协处理器进行渲染工作。在一个复杂的场景中，场景对象的数量可能从几万到几百万。

同 Intel Xeon Phi 协处理器相比，主机处理器在单线程操作上快得多。这对于磁盘 IO 操作也是如此，由于协处理器不能直接访问存储子系统。因此，在主机端加载大量场景数据是符合逻辑的，处理完这些数据之后，我们只需要传输协处理器并行渲染所需要的数据。之后会把渲染图像送回主机，或者写成磁盘上的文件，或者在图形用户界面上显示。一个渲染器的执行流程图如图 20-8 展示。

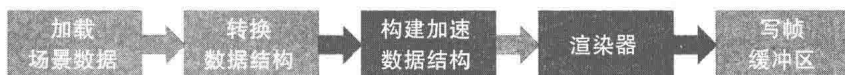


图 20-8 Embree 渲染器执行流程。加载、转换和写入部分的计算都在主机处理器上运行，构建和渲染部分的计算是在协处理器上运行的。指向构建的箭头和从渲染器指出的箭头表示数据传输发生的地方

在 Embree 中，指令参数（通常为小块的数据），使用 COI 管道执行函数的混合数据参数来传输。这需要程序员来定义、串行化，一旦这些数据传送到它们的目的地，反序列化这些数据，以正确地使用它们。

除了传输带宽之外，另一个需要注意的重要度量标准是 COI 执行函数的延迟，即使没有任何缓冲区数据传输，延迟时间也大概会有 0.3ms。当处理大量的几何图形时，如果要一次传输一个几何对象，一百万个对象会占用至少 300s。对于大量的数据传输，如几何图形，重用普通缓冲区来传送大量数据是更有效的。从图 20-6 所示的结果中，我们可以注意到 COIBufferWrite 函数随着数据量的增加而变慢。这是由于基准测试代码是单线程的，并且非一致性访存限制了单核可获得的内存带宽。但是随着数据量的增加，传输带宽提高。如果我们的目标是使得传输大小为 D 的数据所用的时间总量最小（缓冲区写入和传输），我们需要解决一个最小化的问题。设缓冲区的写带宽为 x ，传输带宽为 y ，那么传输大小为 D 的数据所用的总时间 T 为：

$$T = \frac{D(x+y)}{xy}$$

基于图 20-6 中的测量值，一个大小为 16MB 的普通缓冲区在传输数据时会有最佳的性能表现。这仅仅基于简个测试系统配置并且很可能在不同的系统上会有不同结果。关键是正确评估这些指标，并正确评估它们对传输速度的影响，以找到对应用程序最佳的方案。

20.4 总结

本章描述了如何使用 COI 库在主机和协处理器间传输数据，讨论了不同种类缓冲器的利弊，并分析了它们的带宽和延迟性能。那些利用协处理器优势的任何重要的应用程序都会与同主机进行数据通信。编译辅助卸载模型提供便利，而 COI 库提供控制。对于那些开发人员来说，当他们把应用程序在协处理器上取得最佳性能作为目标时，这将是一个重要的工具。本章只覆盖了 COI 库的很少一部分内容。对于库的深入使用感兴趣的读者可以查阅 COI 接口文档的所有选项。

使用 Embree 作为 COI 库在现实世界的应用，我们示范了如何有效地在主机和协处理器端传输不同类型的数据。由于每次调用 COI 执行函数都会有一定的延迟，因此如果没有正确管理和优化，数据传输可能会大大降低应用程序性能。

在这里我要特别感谢 COI 技术主管 Russell McGuire，感谢他的真知灼见和对于 COI 库相关问题的解答。

20.5 更多信息

这里有一些同本章相关的阅读材料：

- Intel 多核平台软件栈 (MPSS), <https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>。
- Embree, <http://embree.github.io>。
- 本章和其他章代码的下载地址 <http://lotsofcores.com>。

高性能光线追踪

Gregory S. Johnson^{*}, Ingo Wald^{*}, Sven Woop[†], Carsten Benthin[†], Manfred Ernst^{*,⊖}

^{*}美国, Intel 公司; [†]德国, Intel 公司

光线追踪是一种用于场景合成的图像生成技术。因为光线追踪模拟现实世界中的物理光传输, 所以它可用来实现高品质甚至具更逼真的结果(见图 21-1)。出于这个原因, 光线追踪通常出现于专业渲染应用中, 如电影制作、建筑渲染, 以及汽车视觉预览等。



图 21-1 通过 Embree 内核构建的光线追踪器产生的图像。龙的表面采用基于物理的反射模型来模拟铜的外观

但是, 光线追踪是计算密集型算法。在单个图像的生产过程中, 数以百万计的光线需要追踪。而且, 在跟踪相邻光线时, 高品质的照明效果(如镜面反射和折射)的计算限制了基于高速缓存的优化价值。

也就是说, 光线追踪是一个高度并行的工作负载。从相同图像的不同像素发出的射线可以彼此独立地追踪, 甚至任何单一射线的有些操作可以并行执行。原则上, 光线追踪非常适合现代多核向量并行芯片架构。在实践中, 对于程序员一个关键的挑战是有效地将光线追踪的内在并行性映射到芯片的功能单元上。

本章描述了一个名为 Embree 的开源光线追踪框架。在专业的渲染应用中, 几何形状复杂的场景和高品质的照明效果都是很常见的。Embree 旨在针对专业的渲染应用获得非常好的性能。Embree 由一组加速基本光线追踪运算的低层计算内核组成。这些内核通过组合使用多线程以及特定指令集架构(ISA)的向量化, 最大化地利用现代的 x86 架构。通过一个高层 API, 可以以最小的编码量来将这些内核用于现有的渲染应用(渲染器)中。本章展示了光线追踪的概况, 介绍了 Embree 中如何采用并行化方法使其在 Intel Xeon 处理器以及 Intel Xeon Phi 协处理上获得高性能, 并通过示例代码演示了在一个全渲染应用中如何使用 Embree 内核。

⊖ 当前工作单位: 美国谷歌公司。

21.1 背景

光线追踪中最基本的运算即通过给定方向（即一条光线）上给定的点，判断场景中的物体是否可见。这项操作可以用来渲染图像，如图 21-2 所示。照相机通过像平面的每一个像素点发射一条或者多条光线。在每一个击中点，如果表面是反射性的或者半透明的，可能会产生一条辅助光线。该过程会继续递归进行，直到有截止情况发生（例如，光线路径的长度）或者光线不能穿过某个物体。给定像素的颜色是由沿着光线路径的照明情况来决定的，并且是对经像素发出的光线求平均值的结果。

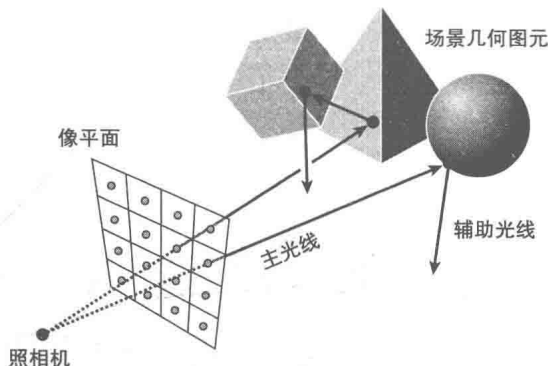


图 21-2 光线追踪中的一种基本操作用于检测与光线相交的物体。一条或多条主光线可以通过像平面中的每个像素进行追踪。在每个击中点会产生辅助光线

此操作的一个简单实现会检测每条光线与场景中每个几何图元的交叉点。然而，包含数以百万计的图元以及数以百万计的光线渲染的场景所需计算量是难以想象的。出于这个原因，场景几何通常存储在分层的空间数据结构内，例如一个受限容量层次结构（BVH）（见图 21-3）。对每条光线与数据结构的根节点（即全局场景边界）的交叉点进行检测，然后检测由光线遍历的场景的受限子区域的子节点。实际场景几何存储在叶节点中，这也是执行光线几何图元交叉点检测的地方。使用这样的数据结构显著降低了生成图像时交叉点检测的总次数，但是增加了代码并行化的复杂度。

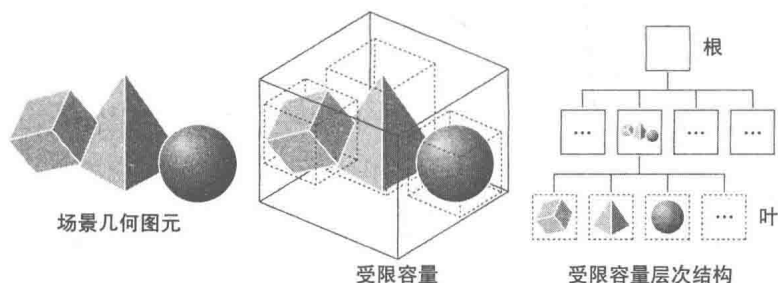


图 21-3 受限容量层次结构（BVH）可以减少光线几何图元交叉点检测的次数。BVH 中的每个节点存储了其子节点所包含空间区域的并集，场景几何图元存储在子节点。在场景中互相接近的物体存放在 BVH 中的邻居节点中

21.2 向量化的光线遍历

在任务级别，光线追踪是很容易平行的。与图像中不同像素点相关联的光线可以独立

地追踪。像素块通常通过不同线程渲染，而分块大小以及线程分配应尽可能达到负载均衡，同时保护相同分块中邻居光线之间的空间相关性。

但是，使光线追踪算法有效地利用现代 CPU 处理器的向量单元是非常困难的。例如，在层次化的空间数据结构下，光线的行进需要细粒度的数据依赖分支以及非规则内存访问模式，这妨碍了自动向量化。更糟的是，将这样的内核以最优的方式映射到特定 ISA 下，即使对于有经验的程序员也并非易事。

通过将不同的光线分配到不同的向量通道且通过 BVH 追踪光线“束”，对光线遍历进行向量化是可能的。因为光线束追踪是不依赖于向量单元宽度或 ISA 的，所以它可以广泛应用并且可以在空间相关光线（即具有相似起点以及方向的光线）上获得很高的向量单元利用率。然而，对于非空间相关光线（低向量单元利用率），光线束追踪效果相对较差，同时必须对渲染器并行化，以便可以同时追踪多条光线。

或者，对于单条光线，空间数据结构的遍历也可以向量化（称为“单光线向量化”）。例如，多分支 BVH 结构可以实现多个节点或几何图元与同一光线交叉点的并行检测。对于每个节点拥有 4 个子节点的 BVH 数据结构，可以很好地使用宽度为 4 以及 16 的向量单元，但较高的分支因数降低了优化效果。

一般来说，对于非相干光线，单光线的向量化速度比光线束追踪要快，并且可以在标量渲染器中使用，而在相干光线束中情况却相反。出于这个原因，一种在两者之间动态切换的混合技术正在发展，即在光线相关时采用包追踪，非相干时则切换到单光线的向量化。

21.3 Embree 光线追踪内核

开发 Embree 的动力来自于以下 4 种观点。第一，全功能的渲染器可以由一小组常用的光线跟踪操作（例如，BVH 建立和遍历）建成。第二，虽然 CPU 架构原则上适用于其中具有丰富的细粒度数据相关分支（例如，分层数据结构的遍历）的计算密集型工作负载，但是实现高吞吐量实际上是极具挑战的。第三，射线追踪广泛用于专业渲染中，并且有必要在这些应用中实现高性能。第四，近期的很多关于光线追踪加速的研究工作还没有找到切实可行的方法，这是由于将这些技术集成到现有渲染器的复杂性以及特殊性导致的。

基于上述观点，Embree 针对 x86 架构实现了一组低级别高性能内核，主要面向 BVH 结构、光线遍历，以及光线三角形交叉点。这些内核利用了多个层次的并行性，其中的重点是高效的向量化技术以及最新的优化技术。在不同的工作负载以及指令级架构上获得高性能需要对每个内核提供不同版本的代码。

在 Intel Xeon Phi 协处理器上，光线束追踪（针对空间相干光线）和单光线遍历（针对非相干光线）均支持，同时支持两种方法在运行时的动态切换。在追踪光线束时，16 条光线同时追踪。与此相反，在 BVH 的单一光线遍历中，关于该光线的 4 个包围盒并行检测，并且交叉点检测的每个坐标轴（即 x 、 y 、 z ）可以并行计算。类似地，在 BVH 子节点的单一光线交叉点中，关于该光线的 4 个三角形可以并行检测，并且每个交叉点的 3 个坐标可以同时计算。

在一个应用程序中，Embree 内核可以通过高层次的 API 来访问，这屏蔽了数据结构内存布局以及 ISA 优化等实现细节。这些内核依次由一组通用组件创建，其中这些组件对原子操作、向量并行操作、同步以及多线程等进行了跨平台封装。通过这一层，可以对目前所有的 x86 体系结构，操作系统（Linux、Microsoft Windows、苹果 Mac OS），以及编译器（Intel C++ 编译器、GCC、Clang，以及 Microsoft Visual Studio）进行支持。

21.4 在应用程序中使用 Embree

Embree 内核可以很容易使用在现有的渲染器中。但是（如在许多应用中），在渲染器的性能和并行化程度之间具有三方面折中。Embree 单光线的 SIMD 内核可以使用在标量渲染器中，而且在向量宽度为 4 时往往能取得良好的利用率。然而，单光线方法在向量宽度为 16 的单元（例如，Intel Xeon Phi 协处理器）上效果较差。为了获得高利用率，光线束与单一光线的混合技术是必需的，并且需要渲染器可以并行产生多条光线。

在 Intel Xeon 处理器以及 Intel Xeon Phi 协处理器上，有几种方法可以实现一个并行渲染器。Intel 的 SPMD 程序编译器（ispc）是其中一种选择。图 21-4 给出了一段 ispc 代码，该代码实现了一个可以并行发射多条光线的简单渲染器（见图 21-5）。ispc 采用了 C89 的大部分语言规范，以及 C99 和 C++ 的部分特性，从语言级支持任务并行，并通过关键字支持数据并行。在不同的 ISA 以及向量宽度上代码无须重写，这是 SPMD 编程模型的一个关键优势。

```

/* Render a tile of pixels in the image. */
task void renderTile(uniform int* uniform pixels,
                    const uniform int imageWidth,
                    const uniform int imageHeight,
                    const uniform Camera& camera)
{
    /* Tile indices. */
    uniform int tileCountX = (imageWidth + TILE_WIDTH - 1) / TILE_WIDTH;
    uniform int tileIndexY = taskIndex / tileCountX;
    uniform int tileIndexX = taskIndex - tileIndexY * tileCountX;

    /* Tile extents in image coordinates. */
    uniform int x0 = tileIndexX * TILE_WIDTH;
    uniform int y0 = tileIndexY * TILE_HEIGHT;
    uniform int x1 = min(x0 + TILE_WIDTH, imageWidth);
    uniform int y1 = min(y0 + TILE_HEIGHT, imageHeight);

    /* Render pixels across SIMD lanes. */
    foreach (y = y0 ... y1, x = x0 ... x1) {
        Vec3f color = renderPixel(x, y, camera);
        unsigned int r = (unsigned int) (255.0f * clamp(color.x, 0.0f, 1.0f));
        unsigned int g = (unsigned int) (255.0f * clamp(color.y, 0.0f, 1.0f));
        unsigned int b = (unsigned int) (255.0f * clamp(color.z, 0.0f, 1.0f));
        pixels[y * imageWidth + x] = (b << 16) + (g << 8) + r;
    }
}

/* Called by the C++ code to render an image. */
export void renderImage(uniform int* uniform pixels,
                      const uniform int imageWidth,
                      const uniform int imageHeight,
                      const uniform Camera& camera)
{
    /* Image dimensions in tiles. */
    uniform int tileCountX = (imageWidth + TILE_WIDTH - 1) / TILE_WIDTH;
    uniform int tileCountY = (imageHeight + TILE_HEIGHT - 1) / TILE_HEIGHT;
    uniform int tileCount = tileCountX * tileCountY;

    /* Render tiles of pixels on different threads. */
    launch[tileCount] renderTile(pixels, imageWidth, imageHeight, camera);

    /* Wait until all tiles have been rendered. */
    sync;
}

```

图 21-4 ispc 代码实现了一个可以使用 Embree 光线追踪的简单渲染器。该计算是多线程和向量化的

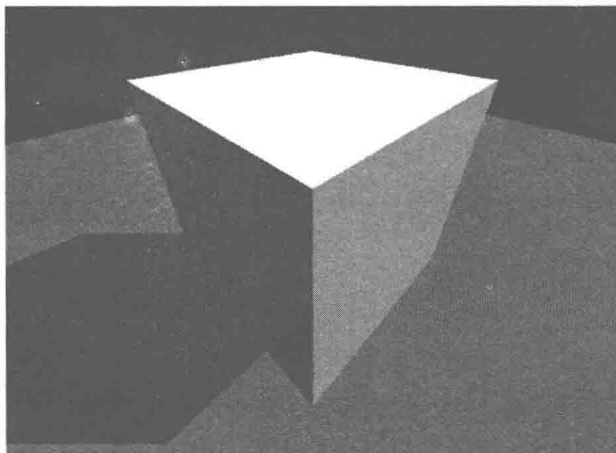


图 21-5 一个使用了 Embree 光线追踪内核的样例程序的输出结果。该应用中渲染器部分的代码见图 21-4 以及图 21-6

```

/* Compute a color at the hit point of the ray. */
Vec3f shadePixel(const RTCRay& ray) {
    /* Look up the color associated with the hit primitive. */
    Vec3f color = colorLookupTable[ray.primID] * 0.5f;

    /* A vector in the direction of the simulated light source. */
    Vec3f lightVector = normalize(Vec3f(-1.0f, -1.0f, -1.0f));

    /* Cosine of the light vector and surface normal for diffuse shading. */
    float weight = clamp(-dot(lightVector, normalize(ray.Ng)), 0.0f, 1.0f);

    /* Initialize a shadow ray. */
    RTCRay shadowRay;
    shadowRay.org = ray.org + ray.tfar * ray.dir;
    shadowRay.dir = neg(lightVector);
    shadowRay.tnear = 0.001f;
    shadowRay.tfar = inf;
    shadowRay.geomID = RTC_INVALID_GEOMETRY_ID;
    shadowRay.primID = RTC_INVALID_GEOMETRY_ID;

    /* Determine if the original hit point is occluded. */
    rtcOccluded(sceneID, shadowRay);

    /* Add a diffuse component if the hit point is not in shadow. */
    return((shadowRay.geomID >= 0) ? color : color + color * weight);
}

/* Render a pixel in the image. */
Vec3f renderPixel(float x, float y, const uniform Camera& camera) {
    /* Initialize a primary ray. */
    RTCRay ray;
    ray.org = camera.position;
    ray.dir = normalize(x * camera.vx + y * camera.vy + camera.vz);
    ray.tnear = 0.0f;
    ray.tfar = inf;
    ray.geomID = RTC_INVALID_GEOMETRY_ID;
    ray.primID = RTC_INVALID_GEOMETRY_ID;

    /* Find the nearest object in the scene hit by the ray. */
    rtcIntersect(sceneID, ray);

    /* Compute shading if an object in the scene was hit by the ray. */
    return((ray.geomID >= 0) ? shadePixel(ray) : Vec3f(0.0f, 0.0f, 0.0f));
}

```

图 21-6 通过 Embree API 调用 Embree 内核实现的用于渲染简单阴影下单一像素的 ispc 代码

图 21-6 展示了一段 ispc 代码，该代码可以利用 Embree API 来追踪一条特定光线。Embree API 包含如下功能：几何图元定义，在几何图元上构建 BVH（并未在这里展示），发布交叉点以及阻塞查询。尽管使用 API 不是必需的，但它可以简化应用程序的开发。重要的是，API 隐藏了技术细节，例如在特定的几何图元以及 ISA 下，哪些内核和内存中的数据布局可以获得最佳性能。该 API 假设内核以及应用程序在相同的地址空间中。

21.5 性能

图 21-8 展示了 Intel Xeon Phi 7120 协处理器（61 核，1.28GHz）上的光线遍历速率，其中对直接以及间接照明的 3 种渲染器以及图 21-7 所示的 4 种场景进行了性能对比。性能展示包括如下三方面：对基于 C++ 实现的基于标量光线追踪的标量渲染器，使用 Embree 单光线 SIMD 内核的标量渲染器，以及利用 ispc 以及 Embree 光线束 / 单光线 SIMD 内核的混合技术实现的并行渲染器。标量渲染器中没有显式的向量化代码，但是也利用了 Embree 中的几个高效类，包括高层次三元组类型（例如，点、颜色）以及相关操作，这些操作可以自动映射到低层的 SIMD 类型以及指令。我们使用了 2.2 版本的 Embree 内核库，并通过 Intel Composer XE 14.0.1 以及 Intel ispc 1.6.0 进行构建。虽然这些结果对基于 Embree 内核库实现的完整渲染器所能达到的性能具有指示作用，但程序员在实际中获得的性能由具体应用以及工作负载而定。

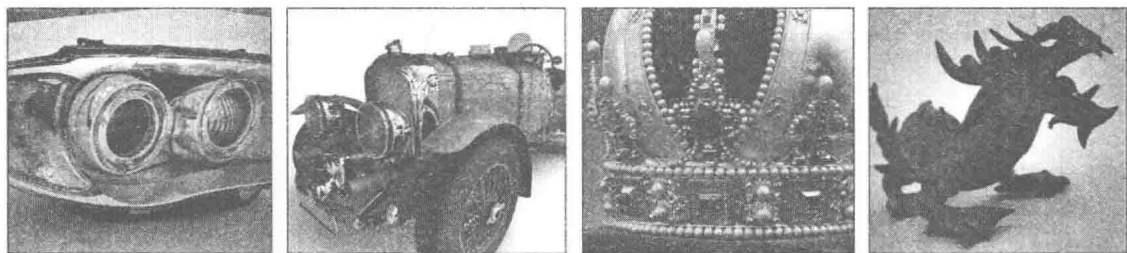


图 21-7 在性能测试中使用的模型：前大灯（80 万个三角形），TurboSquid 的宾利（230 万个三角形），Martin Lubich 的奥地利皇冠（480 万个三角形），以及斯坦福大学计算机图形实验室的龙（740 万个三角形）

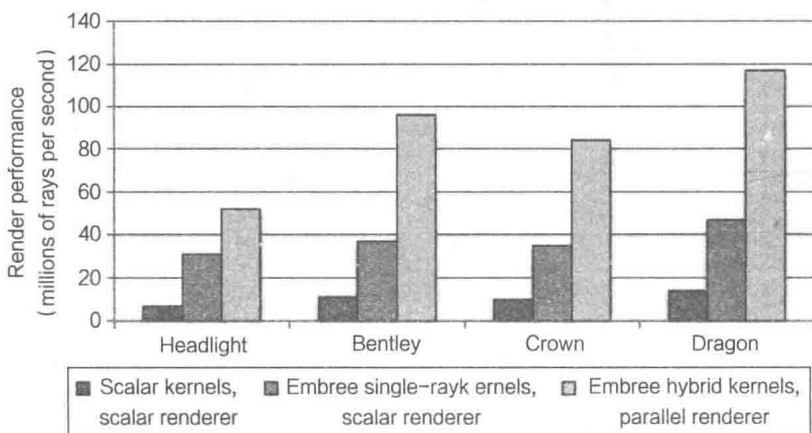


图 21-8 不同级别向量化下三种渲染器的性能（越高越好）。给出的结果是总的光线追踪数除以总帧时间，包括采样（每像素 16 个样本）和阴影（通常为总帧时间的 30% ~ 50%）

向量化在渲染器中可以提升阴影和采样的性能，并且可以使用 Embree 光线束内核以及混合遍历内核。这些结果说明了这两方面的好处，同时也表明完全向量化的渲染器对开发具有宽向量长度的体系结构的计算能力十分重要。在一些情况下，设计限制或所需工作量会使渲染器的向量化不可行。在这些情况下，单光线的 Embree SIMD 内核仍可用于提升光线遍历的性能。

但是 Embree 的性能与最新的图形处理单元（GPU）相比会怎么样呢？图 21-9 对比了一个基于 Embree 混合光线束 / 单光线 SIMD 内核的 ispc 并行渲染器与一个 NVIDIA OptiX 架构上基于 GPU 实现的渲染器的性能。OptiX 简化了 NVIDIA GPU 上高性能渲染器的开发，它集成了低级别内核，一个可编程的光线追踪流水，一个特定领域的编程模型和编译器，以及一个场景图规范（在这里使用了 OptiX 3.5.1 以及 CUDA 5.5）。这两种渲染器都只计算漫反射阴影，从而最小化应用程序特定的计算对渲染时间的影响。在所有情况下，Intel Xeon Phi 7120 协处理器上 Embree 渲染器的表现不逊色于 OptiX 在 NVIDIA GeForce GTX Titan 上的表现。

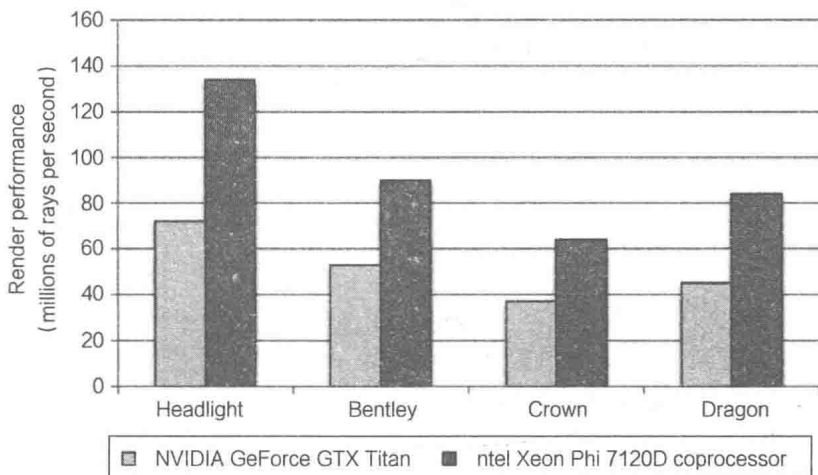


图 21-9 在 NVIDIA OptiX 系统上基于 GPU 实现的并行渲染器的性能（越高越好），以及一个基于 Embree 混合光线束 / 单光线 SIMD 内核的 ispc 并行渲染器

21.6 总结

光线追踪是一种在计算机场景中渲染高质量图像重要且广泛应用的方法。虽然光线追踪是计算密集型的，但它也是一种高度并行的工作负载。因此，光线追踪受益于现代处理器架构，这些结构通过功能单元的并行性（例如，多核）以实现高吞吐量。与图像中不同像素点相关联的射线可以独立地追踪，并且每条光线上的操作通常可以并行执行。这种丰富的并行性可以通过多种方式映射到多核向量处理器和协处理器上。但是，对于程序员，产生最佳性能的映射并不总是显而易见的。例如对于光线束追踪，在光线出现分支（在辅助光线下这是常见情况）时，简单的向量化策略会减少收益。

Embree 通过提供一系列光线追踪通用操作内核来解决这一难题，并对不同的向量宽度、工作负载（例如，相干和非相干光线分布，静态和动态的场景），以及应用特定需求（例如，最大的性能或最小的内存使用量）分别进行调优。通过这些内核构建的渲染器，使 BVH 构建以及光线遍历在 Intel Xeon Phi 协处理器上获得的性能可以媲美（通常高于）任何现有的

CPU 或 GPU。

此外，Embree 内核方法是广泛适用的，并且避免为渲染器带来设计或目标平台上的限制。例如，Embree 内核可使渲染器以最高性能并行追踪多条光线，但并不要求该渲染器是并行的。通过相同的 API，Embree 单光线 SIMD 内核可以用于标量渲染器中。通过对两种类型渲染器在多种 ISA 上的无缝支持，Embree 可以使开发人员从标量渲染逐渐过渡到完全并行渲染，且可以在 Intel 处理器和协处理器之间切换。

21.7 更多信息

下面是 Intel 处理器以及协处理器上关于高性能光线追踪的补充阅读材料：

- 本章示例中对应的完整的源代码：<https://github.com/embree/embree/tree/master/tutorials/tutorial00>。
- Embree 技术概述：<http://embree.github.io>。
- Embree 内核库的源代码：<https://github.com/embree>。
- Embree—a kernel framework for efficient CPU ray tracing. ACM Transactions on Graphics(Proceedings of ACM SIGGRAPH), 2014.
- Intel SPMD 程序编译器：<https://ispc.github.io>。

OpenCL 程序的可移植性能

Simon McIntosh-Smith*, Tim Mattson†

* 英国, 布里斯托大学, † 美国, Intel

用于异构计算平台的现代多核处理器, 都会使用多线程编程模型进行多核编程, 并且每个内核都有向量计算单元。Intel Xeon 处理器是如此, Intel Xeon Phi 协处理器更是如此。每台计算机都是一个异构计算系统, 这是 OpenCL 标准不断完善动力。在传统异构计算中, CPU 和 GPU 使用不同的编程模型。

理想情况下, 编译器能够通过使用循环展开和重构等方法对应用程序自动进行向量化优化, 使最内层的指令块可以映射到处理器的向量指令集上。但实际上, 编译器自动向量化的效果并不是很好。因此, 为提高应用程序性能, 程序员不得不将代码修改为编译器可向量化的形式, 或者使用新编程语言定义的显式向量化结构类型, 甚至直接使用类汇编代码的内部函数。这样产生的代码将对向量指令的宽度非常敏感, 因此具有高度不可移植性。

本章将详细讨论如下假设: 如果将多核处理器系统看作异构计算平台, 那么可通过 GPU 的编程方式对其编程: 将单个编程模型用于多核和向量化编程。更进一步讲, 这些程序也可以很好地运行在 GPU 计算平台上。这样, 应用程序便具有更好的移植性, 从而最大限度地分摊编程成本。

本章使用的编程模型是 OpenCL。首先, 本章使用矩阵乘示例介绍 OpenCL 的关键概念, 并通过精心优化, 使矩阵乘算法在 Intel Xeon Phi 协处理器上的性能达到其峰值性能较高百分比。然后, 本章将介绍一个真实的案例研究: BUDE 分子对接。OpenCL 版本的 BUDE 代码性能可达到 Intel Xeon Phi 协处理器上峰值性能的 32%。同时, 在多个并行硬件平台上 (包括处理器和多个厂商的 GPU), 都可以达到对应峰值性能的较高百分比。

22.1 两难的困境

越来越多的并行计算意味着可以在包含不同处理元件的异构环境中运行应用程序。程序员负责将应用程序映射到 CPU 线程中的标量指令或者向量计算单元的向量指令, 或者将任务块转移到协处理器甚至 GPU 上。每个计算平台都在快速改进, 并提出一个目标: 可以兼容两代产品在架构上发生的根本变化。

应用程序开发者只编写一个简单的串行程序就可以涵盖所有目标计算平台的时代一去不复返了。应用程序开发者编写的程序, 不但能够高效运行在目前所有的硬件平台上, 而且要能够运行在将来的计算平台上。这是因为应用软件的生命周期一般会比硬件平台的生命周期长。程序员不太可能只针对单一硬件平台而忽略其他平台。

这最终会导致一个两难的困境。程序员面临着看似对立的紧张局面: 一方面, 需要针对使用的并行和异构计算平台的特定硬件进行编程; 另一方面, 需要管理软件的复杂性, 在理想情况下, 一个代码库可以支持所有相关计算系统。

OpenCL 可用来解决众核程序员的两难选择。本章将不讨论 OpenCL 是否为异构编程的

最佳选择。程序员可自由选择任意方法来满足自己的需求,而且希望所有程序员都使用同一方法是没有好处的。OpenCL 是第一个广泛使用的并力求涵盖 FPGA、CPU、协处理器、GPU 以及其他加速器的开发标准。对于需要支持如此广泛计算平台的程序员来说,OpenCL 是解决这个两难困境的最佳选择。因此,了解 OpenCL 标准并学会如何使用,是非常重要的。

22.2 OpenCL 简介

OpenCL 是一个解决上节提出的异构计算系统面临的两难困境的行业标准。使用 OpenCL 可以实现“一份代码运行于多个异构系统”的目标。

OpenCL 1.0 于 2008 年下半年发布,到 2009 年年中,已经被多个 CPU 和 GPU 厂商支持。随后,OpenCL 标准经过了快速调整和改进(2010 年发布 OpenCL 1.1,2011 年发布 OpenCL 1.2),并在 2013 年发布了 OpenCL 2.0。OpenCL 的快速发展,对于程序员的与时俱进提出了严峻的挑战。然而,因为计算硬件迅猛的发展速度,这些改进是必需的。

本节将讨论 OpenCL 的核心特征,从 OpenCL 标准发布之日起,这些特征并没有进行大的变动。我们将通过以下几个模型介绍 OpenCL:

- 平台模型
- 执行模型
- 内存模型

OpenCL 平台模型如图 22-1 所示,它由一个主机和一个或者多个设备构成。主机定义为基于 CPU 并支持文件 I/O、用户交互和其他传统笔记本电脑和服务器的基本功能的计算机。设备定义为执行 OpenCL 程序中大块计算任务的计算部件,如 GPU、众核协处理器和其他实现 OpenCL 的专用设备。每个设备由一个或者多个计算单元(Compute Unit, CU)构成,每个 CU 由一个或者多个计算元件(Processing Element, PE)构成。PE 为 OpenCL 程序中最细粒度的计算单元。

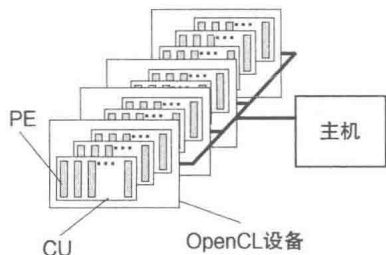


图 22-1 OpenCL 平台模型: 一个主机和多个设备。每个设备有一个或者多个 CU, 每个 CU 有一个或者多个 PE

当进行 OpenCL 程序优化时,平台模型给予程序员一个硬件抽象。然后,通过学习平台模型到不同目标计算平台的映射方式,程序员可以在不牺牲可移植性的前提下优化软件。

OpenCL 程序按照细粒度 SPMD (Single Program Multiple Data, 单程序多数据) 的模式执行。考虑图 22-2a 给出的矩阵乘的串程序: 其中包含一个三层的嵌套循环,通过求矩阵 A 的第 i 行和矩阵 B 的第 j 列的叉积,计算目标矩阵元素 $C(i, j)$ 。

OpenCL 的核心思想是定义 1、2 和 3 维的索引空间。程序员可以将具体问题映射到该索引空间上。同时,将执行主要计算过程的代码块定义为核(在索引空间中每个点运行的代码实例)。

对于矩阵乘代码,我们将最外层的两层循环映射到一个二维索引空间上,定义核运行最内存循环(在 k 上)。然后,我们在索引空间的每个点上运行核实例,在 OpenCL 语法中称为工作项。图 22-2a 和图 22-2b 分别给出了矩阵乘的串行代码和对应的核函数。比较这两段代码可以发现: 串行代码中的最外两层循环被查询索引空间中工作项的索引操作代替。最后,我们在索引空间的每个点都执行该核的具体实例来完成计算。


```
void mat_mul(const unsigned int Order,
             const float *A,
             const float *B,
             float *C)
{
    int i, j, k;
    for (i=0; i < Order; i++)
        for (j=0; j < Order; j++)
            for (k = 0; k < Order; k++)
                C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
}
```

a)

```
__kernel void mat_mul(const unsigned int Order,
                      __global const float *A,
                      __global const float *B,
                      __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    for (k = 0; k < Order; k++)
        C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
}
```

b)

图 22-2 a) 矩阵乘串行代码；b) OpenCL 核中的并行矩阵乘

图 22-3 给出了 OpenCL 程序更详细的执行过程，这个过程总结了 OpenCL 执行模型。全局索引空间（在这个例子中为 16*16），包含了在每个点执行核实例的一组工作项。这些工作项分成了许多与全局索引空间有相同形状的工作项块，称为工作组，工作组覆盖整个索引空间。

逻辑上，同属于一个工作组内的所有工作项一起执行。因此，它们可以在执行过程中同步和共享内存。但工作组间不能执行这样的操作。在单个核实例中，工作组的运行顺序是没有限制的，因此工作组间没有定义同步结构。这个限制对数据共享有重要影响，这将在内存层次模型中进行讨论。

对于习惯多线程编程（如 pthreads、Java 线程等）灵活性的程序员来说，这些同步限制可能看起来比较麻烦。然而，OpenCL 执行模型定义这些限制是有原因的。OpenCL 通常用来对高度数据并行的算法进行高吞吐量计算，并通过创建由准备执行的工作组组成的大型内部工作池实现高性能。为保证设备的每个 CU 被任务完全占用，调度器以流水线的方式

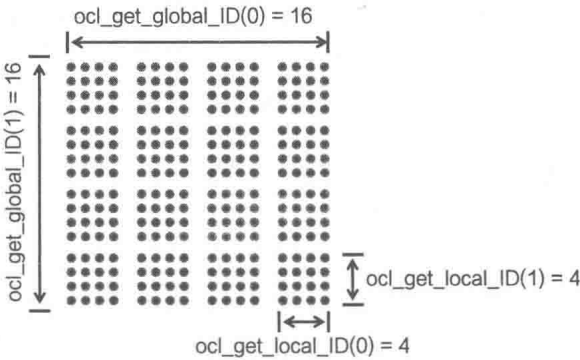


图 22-3 问题被拆解为 $N(N = 1, 2, 3)$ 维索引空间上的点，在 OpenCL 中称为 NDRange。在 NDRange 中每个点上运行的核实例称为工作项。这些工作项分成工作组，从而均匀地把整个索引空间分块

调度这些可运行的工作组。

因为计算设备（如 GPU 和 FPGA）可能会有独立的板上内存，所以异构计算平台不可能提供一个内存一致性的内存空间。因此，OpenCL 中的内存模型，根据平台模型定义了将 OpenCL 内存分解到不同地址空间中的方式，如图 22-4 所示。

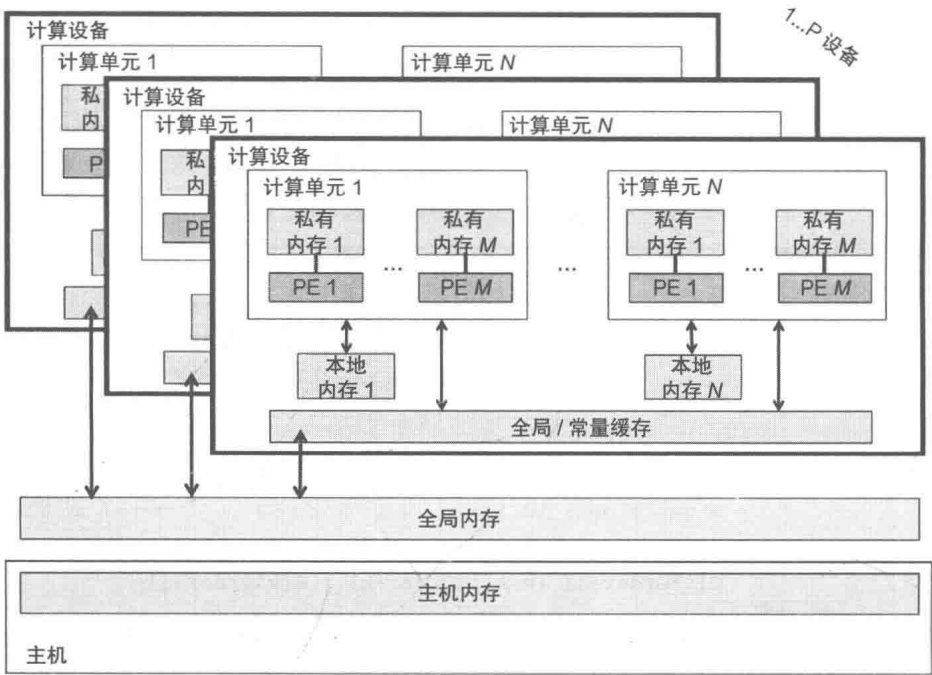


图 22-4 OpenCL 1.X 的内存模型以及与平台模型的对应关系。在一个上下文中有 P 个设备，它们对全局 / 常量内存来说都是可见的

从图 22-4 的底部开始，首先是主机内存，顾名思义，主机内存由主机定义或仅对主机可见（尽管 OpenCL 2.0 已经取消了这个限制）。内存层次架构的下一层为全局内存（global memory），其中包含一块只读内存，称为常量内存（constant memory）。全局内存和常量内存存储 OpenCL 内存对象，并且对于计算（即，程序员定义的上下文）中涉及的所有 OpenCL 设备可见。独立 GPU 或者 FPGA 板上的 DRAM 通常会映射为全局内存。这里值得注意的是，对于独立设备，在主机内存和全局内存之间移动数据需要跨总线（如 PCI-E）传递数据，这可能相对较慢。

在 OpenCL 内部，每个 CU 都有一块称为本地内存（local memory）的本地内存区域。本地内存仅对 CU 内部的 PE 可见，这可非常好地映射到 OpenCL 的执行模型上：一个或者多个工作组运行在一个 CU 上，一个或者多个工作项运行在一个 PE 上。本地内存上的数据可属于一个工作组内的所有工作项共享。OpenCL 存储层次架构的最后部分是私有内存（private memory），它仅对工作项中少量的内存可见。

在 OpenCL 中，不同存储层次上的数据传输都是显式的。也就是说，用户负责主机内存到全局内存的数据传输，其他的也是如此。程序员必须使用 OpenCL API 定义的命令以及内核编程语言来进行主机内存到全局内存、全局内存到本地或者私有内存的数据传输。

22.3 OpenCL 示例：矩阵乘

我们需要借助一个更加复杂的例子来了解这些地址空间，如图 22-5a 描述的矩阵乘核。

虽然这个程序初看起来很复杂，但是基本原理非常简单。我们使用一个众所周知的方法对每个循环（共三个）进行重构，将它们转化为循环对。即第一个循环基于块，第二个循环基于每个块内的元素。这个方法可以让我们通过选择块的大小，将计算需要的工作空间加载到性能更快的内存（即本地内存或私有内存），从而有效提升程序性能。

为了实现这个方法，每个工作项的任务由计算向量点积转变为计算“矩阵点积”。“矩阵点积”的操作数分别为矩阵 A 的行块和矩阵 B 的列块。图 22-5b 给出了该算法的图形化视图。这个算法的每一步都是一个串行矩阵乘（应该是“矩阵点积”——译者注）操作，这就可以对每次数据传输执行更多的计算任务，从而分摊全局内存到片上内存（这里的片上内存指本地内存或私有内存）的数据传输开销。

```
#define blksize 16
__kernel void mmul(
    const unsigned int N,
    __global float* A,
    __global float* B,
    __global float* C,
    __local float* Awrk,
    __local float* Bwrk)
{
    int kloc, Kblk;
    float Ctmp=0.0f;

    // compute element C(i,j)
    int i = get_global_id(0);
    int j = get_global_id(1);

    // element C(i,j) is in
    // block C(Iblk,Jblk)
    int Iblk = get_group_id(0);
    int Jblk = get_group_id(1);

    // C(i,j) is element
    // C(iloc, jloc) inside
    // the block
    int iloc = get_local_id(0);
    int jloc = get_local_id(1);
    int Num_BLK = N/blksize;

    // upper-left-corner and inc for A and B
    int Abase = Iblk*N*blksize;
    int Ainc = blksize;
    int Bbase = Jblk*blksize;
    int Binc = blksize*N;

    // C(Iblk,Jblk) = sum over Kblk of
    // A(Iblk,Kblk)*B(Kblk,Jblk)
    for (Kblk = 0; Kblk<Num_BLK; Kblk++)
    {
        //Load (Iblk,Kblk) and B(Kblk,Jblk).
        //Each work-item loads single
        //elements of the two shared blocks
        Awrk[jloc*blksize+iloc] =
            A[Abase+jloc*N+iloc];
        Bwrk[jloc*blksize+iloc] =
            B[Bbase+jloc*N+iloc];
        barrier(CLK_LOCAL_MEM_FENCE);

        #pragma unroll
        for(kloc=0; kloc<blksize; kloc++)
            Ctmp += Awrk[jloc*blksize+kloc]
                * Bwrk[kloc*blksize+iloc];
        barrier(CLK_LOCAL_MEM_FENCE);
        Abase += Ainc; Bbase += Binc;
    }
    C[j*N+i] = Ctmp;
}
```

a)

图 22-5 a) 基于块的矩阵乘核。这个算法在工作组内计算矩阵 A 行块和矩阵 B 列块的“点积”。因为这些行块和列块都加载进了能够被工作组内所有工作项共享的本地内存中，所以只需要一次从全局内存的数据传输。b) 基于块的矩阵乘算法的图形表示：使用矩阵 A 的行块和矩阵 B 的列块计算矩阵 C 的一块数据。每个工作组映射的全局内存对象的地址为：每个工作组负责处理的矩阵 A 和矩阵 B 数据的基址（ $Abase$ 和 $Bbase$ ）加上对应矩阵子块的地址增量（ $Ainc$ 和 $Binc$ ）。这里为使用图 22-3 定义的 $16*16$ 索引空间的计算方式

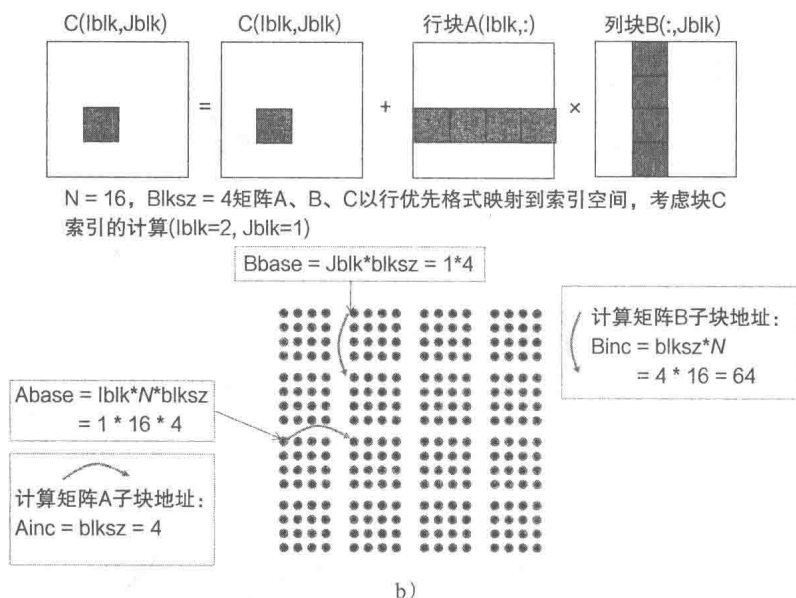


图 22-5 (续)

现在让我们更加详细地了解这个核。首先，核顶部定义了块的大小，从而定义了矩阵 A 、矩阵 B 和矩阵 C 的分块大小 ($blksize$)。选择合适的块大小，从而使矩阵分块能够装载进片上内存 (如，本地内存) 中。然后，开始定义核本身：将函数标记为核，从而使系统知道该函数将要运行在上下文定义的多个类型的设备上。同时，核的参数根据它们所处的地址空间进行标记。例如，原始矩阵位于全局内存上，所以用 “__global” 来标记。除此之外，两个参数标记为 “__local”，以说明这两个参数位于本地内存上，所以这部分数据将会被工作组内的所有工作项共享。

记住，每个工作项将会执行相同核。这是 SPMD 模式的一个例子。每个工作项都有一个唯一全局 ID 和一个本地 ID (工作组内唯一)，工作项将根据这些 ID 进行计算。在核的开始会发现确定工作项全局 ID 和本地 ID 的代码：首先调用 `get_global_id()` 函数确定工作项在整个索引空间的坐标。这个坐标将定义了每个工作项所要负责计算的目标矩阵元素；然后调用 `get_local_id()` 函数，确定工作项在工作组内的坐标。图 22-3 说明了全局空间为 (16,16) 且本地空间为 (4,4) 的一个例子，每个工作项都会有一个从 (0,0) 到 (15,15) 的全局 ID 和一个从 (0,0) 到 (3,3) 的本地 ID。

Subsequent A blocks by shifting index by 计算矩阵 A 子块地址增量：

在图 22-5a 中第二列代码的顶部，分别计算矩阵子块在全局内存中的地址偏移量和地址增量。图 22-5b 展示了一个计算例子 (使用图 22-3 定义的 16×16 索引空间)。

负责核主要计算部分的主循环也遵循上述索引计算。主循环 (基于 $Kblk$) 负责处理矩阵 A 的行块和矩阵 B 的列块。在循环开始，每个工作项并行将数据从全局内存加载到本地内存上。循环随后的代码将会对这些数据进行计算，所以程序员必须在数据加载后面添加栅栏进行同步。该栅栏使用一个本地内存 `fence` 确保栅栏操作之后，工作组内的所有工作项都有一致的内存视图。如果没有栅栏，在其他工作项没有完成数据加载之前，可能会有工作项从本地内存中读取数据。注意，栅栏只对同属于一个工作组的工作项进行同步操作，不同工作组间可以异步执行。

当数据加载完成之后，基于块本地索引 ($kloc$) 的循环进行相关数据计算。这个循环之

后还有一个栅栏。这个栅栏非常重要，因为它确保了只有同属于一个工作组的所有工作项都完成这个循环后，才能继续下次循环迭代的处理。这是很关键的，因为在所有工作项（属于同一个工作组）都完成本地迭代的计算前，我们不想有任何一个工作项继续执行基于 Kblk 循环的下次迭代并进行数据加载工作。

22.4 OpenCL 与 Intel Xeon Phi 协处理器

OpenCL 程序员通过了解平台模型映射到特定系统上的方式来推导性能。例如，数据并行循环是如何转化为核并映射到设备上的，数据如何在内存层次架构上进行传输。考虑图 22-1 介绍的平台模型。SPMD 架构（如 GPU）上，比多指令多数据（Multiple Instruction Multiple Data, MIMD）架构（如 Intel Xeon 处理器和 Intel Xeon Phi 协处理器）在硬件上有更多的限制。作为一个开放标准，OpenCL 的目标是在 SPMD 和 MIMD 架构甚至 FPGA 和 DSP 等差别更大的架构上都能进行高效的并行编程。

OpenCL 在 CPU 平台或者在众核平台（如 Intel Xeon Phi 处理器）上将产生有趣的情况。这两个平台比 GPU 协处理器更接近众核多处理器。将 OpenCL 映射到这些系统上有很多方式。并且映射方式是可以改变的，因为这种选择在硬件而不是在软件。

目前，OpenCL 平台模型通过如下方式映射到 Intel Xeon Phi 协处理器上：CU 映射到内核上的硬件线程，处理器上的内核共享向量单元；PE 映射到向量单元中的通道（lane），因此 OpenCL PE 的数量依赖于计算中使用的数据类型。对于单精度的矩阵乘核，32 位的浮点数据类型使每个向量单元支持 16 个 PE（Intel Xeon Phi 的向量长度为 512。——译者注）。

为了在 Intel Xeon Phi 协处理器上实现高性能，核执行时，所有 PE 必须执行一组一致的工作项控制流（一个工作项运行在向量单元的一个通道上）。虽然 Intel Xeon Phi 向量单元支持条件指令或者工作组大小不是向量单元通道数整数倍的情况，但这样会导致一部分 SIMD 通道空闲和低性能。除此之外，为了实现最佳性能，工作组的数目最好是硬件线程（CU）的整数倍。

考虑一个 60 核的 Intel Xeon Phi 协处理器。每核有 4 条硬件线程和一个 512 位的向量单元（16 个单精度浮点数的宽度）。当全力运行 OpenCL 时，需要开启 $60 \times 4 \times 16 = 3840$ 个工作项。因为每个工作组运行在一个核上，所以工作组的大小应该是 16（向量单元宽度 / 浮点数据类型大小）或者 4×16 （硬件线程数目乘以向量单元中的通道数）。

目前，对于 OpenCL 在 Intel Xeon Phi 协处理上的实现，主机程序运行在 CPU 上，OpenCL 核运行在协处理器上。因此，初始数据存储在 CPU 的 DRAM 上，并且需要将数据传输到协处理器的内存上。当程序开始运行时，针对 Intel Xeon Phi 协处理器的 OpenCL SDK（Software Development Kit）定义了分配在协处理器上的一个内存工作池。对于矩阵乘核，我们发现确保这个工作池足以容纳计算所需矩阵是非常重要的。如果这个工作池过小，系统将会在运行时对工作池进行动态增加，因此增加了从主机到协处理内存的数据传输开销。工作池的大小可以通过设置合适的环境变量来调整：

```
CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB
```

本章最后给出了针对 Intel Xeon Phi 协处理器 OpenCL SDK 详细信息的链接。

22.5 性能评估

图 22-6 显示了矩阵乘的性能结果。我们分别在三个不同平台上进行了性能测试：一个笔记本级别的 CPU（双核 Intel Core I5 处理器 @ 2.5 GHz），一个 GPU（Intel HD graphics 5200

GPU)，一个 Intel Xeon Phi 协处理器。这里值得注意的是，我们的目标不是比较这三个不同类型处理器的性能，而是说明 OpenCL 在不同架构处理器上的可移植性。矩阵规模设置为 1000*1000。在进行性能分析、暴露系统管理内存移动方式（相对于计算）的缺点时，这个矩阵规模是比较小的。当矩阵乘程序的串行版本使用“-O3”编译选项时，在一个笔记本电脑级别的 CPU 内核上可达到 200MFLOPS 的性能。使用“-O3”编译选项，编译器将自动进行向量化优化。在同一个 CPU 上运行矩阵乘算法的 OpenCL 初级实现版本（见图 22-2b），可实现 800MFLOPS 的性能，此时 OpenCL 代码将使用 CPU 的两个内核。相同的 OpenCL 程序在 Intel Xeon Phi 协处理器上可实现 13.6GFLOPS 的性能。基于块的矩阵乘算法（见图 22-5a）实现了性能的大幅提升。我们将块的大小设置为 16，仅仅将程序重新编译一下，就在 CPU、GPU 和协处理器三个计算平台上分别实现了 12GFLOPS、38GFLOPS 和 74GFLOPS 的性能。同目前手工优化实现的数学库提供的性能数据不同，该程序没有使用任何汇编代码。

	处理器	GPU	协处理器
串行C程序 (代码来自图1-2A)	0.2		
OpenCL: 每个C (i, j) 开启 一个工作项……,所有全局内存 (代码来自图1-2B)	0.8		13.6
基于块的并行算法 (blksz=16) (代码来自图1-5A)	12	38 (53)	74 (126)

图 22-6 矩阵规模为 1000*1000 时矩阵乘算法的性能，性能单位为 GFLOPS。1) 处理器：Intel Core i5-2520M CPU @ 2.5 GHz (双核) Windows 7 64 位 OS, Intel 64 位编译器 v13.1.1.171, OpenCLSDK 2013; 2) GPU：Intel Core i7-4850HQ @ 2.3 GHz，集成 Intel HD Graphics 5200 w/ 高速内存, ICC 2013 sp1 update 2 ; 3) 协处理器：Intel Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB。GPU 和协处理器的第二个性能数据是将程序连续运行多次时第二次的性能数据。注意，相对于表中的数据，我们观察到的性能误差要小得多，只不过在表中没有显示

当使用 GPU 和 Intel Xeon Phi 处理器时，数据传输开销非常明显。当我们测试矩阵乘算法的运行时间时，包含了将数据从主机内存传输到全局内存的开销。这就意味着我们包含了通过 PCIe 总线进行数据传输的开销。为了得到这些数字，运行了两次，只给出了第二次的速度。图 22-6 所示括号中的性能数据是将矩阵乘程序连续运行两次时第二次的运行时间。换一句话说，第二次计算可以利用第一次计算用到的数据，即没有通过 PCIe 总线进行数据传输的时间开销。性能数据差异一点也不奇怪，因为当将 Intel Xeon Phi 处理器定位为协处理器时，就要通过 PCIe 总线进行数据传输，这对性能有非常大的影响。下一代 Intel Xeon Phi 产品 Knights Landing 将能够作为系统中的处理器或者计算节点进行操作，这将消除协处理器的数据传输开销。

从具体示例中可以发现，OpenCL 核本身比较简单。OpenCL 的大部分复杂性来自于本章没有给出的主机程序。运行在主机端的程序选择用于计算的设备，管理全局内存，并通过命令队列调度核的执行。一个传统的主机程序主要执行如下步骤：

- 定义平台，平台 = 设备 + 上下文 + 命令队列；
- 创建并编译程序（核函数的动态库）；
- 创建内存对象；

- 定义核（向核函数传递参数）；
- 提交命令：传递内存对象，执行核。

虽然主机程序并不难写，也不难理解，但是很麻烦。本章将不继续讨论主机程序。关于 OpenCL API 以及如何使用它们编写主机函数，本章最后将给出对应参考文献。

22.6 案例研究：分子对接算法

本节给出一个关于 BUDE (Bristol University Docking Engine, 英国布里斯托尔大学的对接引擎) 的案例研究, 说明在实际应用场景中使用 OpenCL 开发性能可移植应用程序的有效性。BUDE 是一段基于分子动力学的代码。Richard 博士和来自英国布里斯托尔大学的研究团队已经开发 BUDE 很多年。BUDE 采用了一个新颖的基于原子-原子的自由能力场来精确预测两个原子相互作用产生的相对约束自由力。这个能力意味着 BUDE 能够用来解决三个不同问题: 1) 虚拟筛选, 通过将数百万个小原子和目标蛋白质对接完成筛选 (见图 22-7); 2) 结合位点检测, 通过使用配体检测蛋白质表面, 完成结合位点的检测 (见图 22-8); 3) 蛋白质与蛋白质对接, 通过系统扫描两个蛋白质的表面, 完成对接 (见图 22-9)。

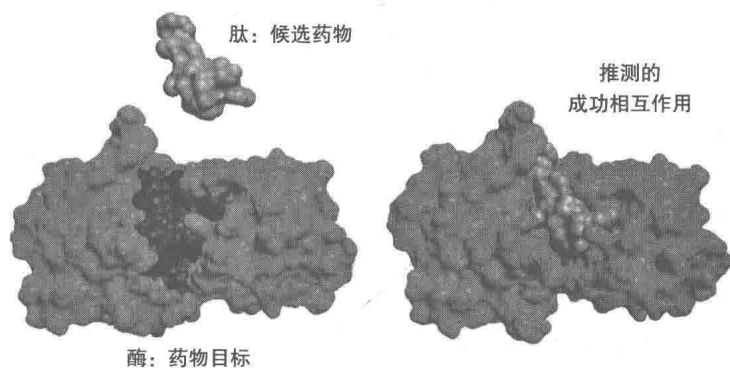


图 22-7 BUDE 通过对接潜在的药物分子 (配体) 和目标 (蛋白质) 实现虚拟筛选

为了在 CPU、GPU 和加速器等不同架构的硬件平台上实现性能可移植, BUDE 已经开发了 OpenCL 版本。目前, BUDE 已经完成了 OpenCL 移植。这就意味着只需要开发和维护一份源代码, 就可以在 CPU、GPU、Intel Xeon Phi 协处理器等其他计算平台上实现非常高的性能。

良好的数据并行性是 BUDE 一个非常重要的特征: 需要执行很多不同分子与分子间的测试, 并且它们之间的运算都是相互独立的。同时, BUDE 属于计算密集型而不是访存密集型算法。这些特征使得 BUDE 算法在高度并行架构上的优化非常简单。每次分子与分子间的测试都会有一个涉及分子位置和方向的独特配置。在 BUDE 术语中, 这个配置称为 pose。

BUDE 在 OpenCL 移植中采用的优化方法遵循如下性能可移植法则。

1. 最大限度开发并行性。OpenCL 程序开发的主要目标就是尽可能开发算法并行性, 并

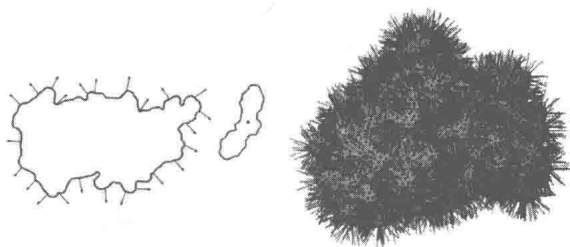


图 22-8 BUDE 通过使用目标配体 (左图) 扫描目标蛋白质表面完成结合位点检测。确定的潜在结合位点如右图所示

尽量减少相互依赖（控制和数据）。对于 BUDE 算法，这意味着需要在较粗的粒度上开发并行性，将不同的 pose 分配给不同的 OpenCL 工作项进行并行计算。这里有比包含在每个分子中的原子数目还多的 pose，因此，pose 级并行可以提供应用程序中的高并行性。

2. 使用众所周知的优化方法。一旦应用程序的并行性得到了充分开发并表现在 OpenCL 程序中，下一步的主要工作就是使用标准源代码优化方法来提升应用程序在众核处理器（如 Intel Xeon Phi 协处理器和 GPU）上的性能。这些优化方法在相关文献中都有很好的解释，例如：为设备内存系统确保良好的访存模式（如合并访存）；尽可能消除控制分支；确保提供可被编译器使用的指针的所有相关信息（如使用 `const` 和 `restrict` 等关键字）。

3. 跨多个设备测试。在开发过程中，代码会在多个设备上反复测试，这些设备至少包括：一个多核多 CPU 系统、一个 Intel Xeon Phi 系统和一个 GPU 系统。保留能够在所有计算平台上提升性能的优化方法。研究在任意一个设备上造成明显性能降低的优化方法。如果是因为设备相关原因导致性能下降，该优化方法将弃用；如果性能下降是由于特定设备或者编译器的 bug，我们将同相关厂商一起解决这个问题。当这个问题解决后，优化方法就会集成到代码中。值得注意的是，在目标设备上几乎所有优化方法都提升了应用程序的性能，而只有一小部分优化方法虽然在一个设备上提升了性能，但在其他设备上会导致性能的明显下降。

4. 使用最极限设备驱动开发。在优化过程中，我们通常会集中精力集中在列表中的绝大多数并行设备上，如 Intel Xeon Phi 协处理器或者高端 GPU。因为这些设备需要最有效的并行代码设计。同时，我们发现当主要精力集中在这些设备上时，相同代码在次并行（Less parallel）设备（如多核 CPU）上的性能也得到了提升。

5. 避免与平台相关的优化方法。许多潜在的优化方法需要利用硬件的相关特性，如内存层次架构中每层的内存大小（在 OpenCL 术语中，如私有内存、本地内存、全局内存和主机内存），工作项分组执行的大小（例如，AMD 的 wavefront 包含 64 个工作项，NVIDIA 的 warp 包含 32 个工作项——译者注）。通常情况下，这些优化方法在特定计算平台上只能获得有限的性能提升，但在其他所有计算平台上都会导致性能下降。对于 BUDE 算法，我们避免使用所有这些与设备相关的优化方法，而是将精力集中在参数调整方面，如能够适用于所有设备的工作组大小（例如，当工作组大小为 32、64 或者 128 等 2 的较小次幂时，在所有设备上的性能经常是最优或者是近似最优的）。

BUDE 是非常复杂的科学程序，进行原子-原子操作的内层循环大约为 50 个基本操作。其中 20% 为条件分支，其他操作为单精度浮点指令（例如，加、乘、倒数和平方根操作），或者加载、存储指令。条件分支是依赖数据的，也就是说，不同工作项很有可能会执行不同的分支。在面向吞吐量的处理器（如 GPU——译者注）上，这种程序特征会导致性能降低。所以，在优化过程中，大量的精力花费在使用语义上等价的线性断言代码（predicated code）代替依赖数据的条件分支语句。断言执行（predicated execution）是一种代替条件分支的方法，该方法执行一个线性指令序列，其计算结果根据条件判断可能会被忽略。使用线性

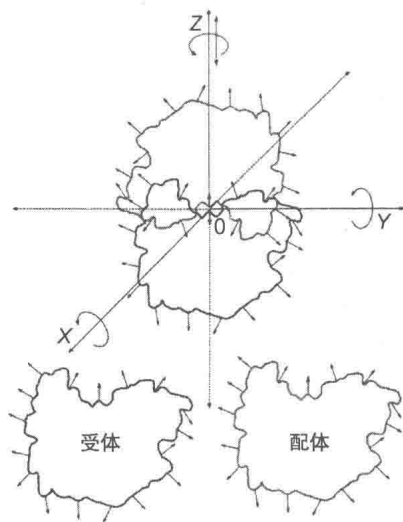


图 22-9 BUDE 通过系统地扫描两个蛋白质的表面，完成蛋白质与蛋白质的对接

断言代码代替条件分支语句具有双重优势：首先条件分支越少意味着在众核架构上的执行流水线就越简单，线程被挂起的次数就越少；其次，减少条件分支可缩短关键路径的长度。在 NVIDIA GTX 680 GPU 上，该优化方法取得了 54.1% 的性能提升。我们本可以消除最内层循环中的所有条件分支，并用线性、断言代码代替。这在所有的测试平台（包括 Intel Xeon Phi 协处理器、Intel Xeon 处理器、NVIDIA GPU 和 AMD GPU）上都会取得明显的性能提升。除此之外，还有一个经常会使用的优化方法：将 OpenCL 核程序修改为对编译器友好的代码。这可以让编译器生成断言代码而不是显式的条件分支。图 22-10 和图 22-11 显示了代码修改前后的不同。

```
if (a > b)
{
    accumulator += (a - b*c);
}

setp.gt.f32 %pred, %a, %b
@!%pred bra $endif
mul.f32 %t0, %b, %c
sub.f32 %t1, %a, %t0
add.f32 %accumulator, %accumulator, %t1
$endif:
```

图 22-10 在生成线性代码时，在修改为对编译器友好的代码之前，BUDE 中的典型代码段。编译器生成的汇编指令包含一个依赖数据的条件分支（“bra”指令）

```
temp = (a - b*c);
mask = (a > b ? 1 : 0);
accumulator += (mask * temp);

mul.f32 %t0, %b, %c
sub.f32 %temp, %a, %t0
setp.gt.f32 %pred, %a, %b
selp.f32 %mask, %one, %zero, %pred
mad.f32 %accumulator, %mask, %temp, %accumulator
```

图 22-11 将图 22-10 中的代码转变为语义相同但对编译器友好的代码。编译器生成的汇编指令为线性断言代码。这在绝大多数架构上会取得更佳性能

22.7 性能评估：性能可移植性

BUDE 是计算密集型程序，通过使用本章描述的优化方法，我们实现了一个具有良好分支特性（最大限度地减少条件分支，当条件分支不可避免时，尽量减少不同的分支）和良好访存模式的高度并行版本。最终，BUDE 在所有的测试设备上都实现了较高的单精度浮点数性能。

当在 Intel Xeon Phi 协处理器和 Intel Xeon CPU 上运行相同的 OpenCL 代码时，单精度浮点运算性能分别达到了 681GFLOPS（Intel Xeon Phi SE10P 协处理器，61 核，1.1GHz）

和 346GFLOPS (两路 Intel E5-2687W 8 核 CPU, 3.1 GHz, 共 16 核)。这是整个 BUDE 程序的运行性能, 而不仅仅是一个内层循环或者内核的性能。BUDE 程序在两个计算平台上分别达到了峰值浮点运算性能的 32% (Intel Xeon Phi SE10P 协处理器) 和 44% (16 核 Intel Xeon 处理器系统)。其中, 计算平台的峰值浮点运算性能是指浮点运算的 SIMD 性能。从中可以看出, Intel 的 OpenCL 实现在 Intel Xeon Phi 协处理器和 Intel Xeon 处理器上都可以对 OpenCL 代码有效地向量化。相对于 Intel Xeon 处理器系列的高端 16 核 Sandy Bridge 架构, BUDE 在 Intel Xeon Phi 协处理器上实现了 1.94 倍的性能提升。性能结果如图 22-12 所示。

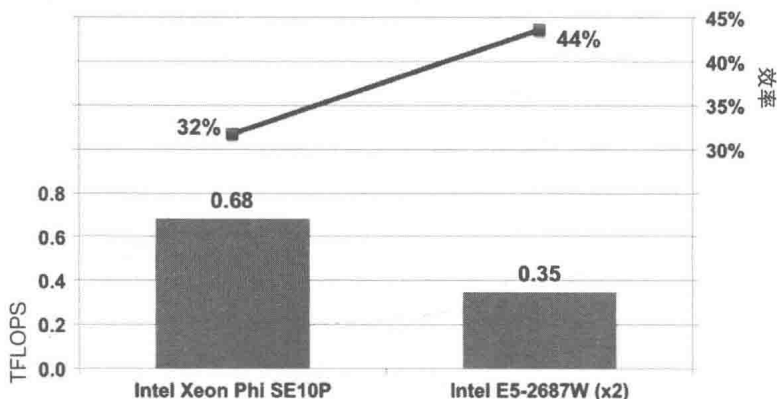


图 22-12 BUDE 在处理器和协处理器测试系统上的性能

OpenCL 程序的运行性能接近于计算平台的峰值性能很有意义, 并且证明了 OpenCL 在不依赖于复杂的运行时代码生成或者源代码自动调优机制的前提下, 具有开发性能可移植的单源应用程序的潜力。

值得注意的是, 当同一份 BUDE 代码编译后运行在不同的众核平台上 (如 AMD GPU 和 NVIDIA GPU) 时, 也会实现类似的高性能。图 22-13 显示了 BUDE 代码在不同计算平台上的性能。

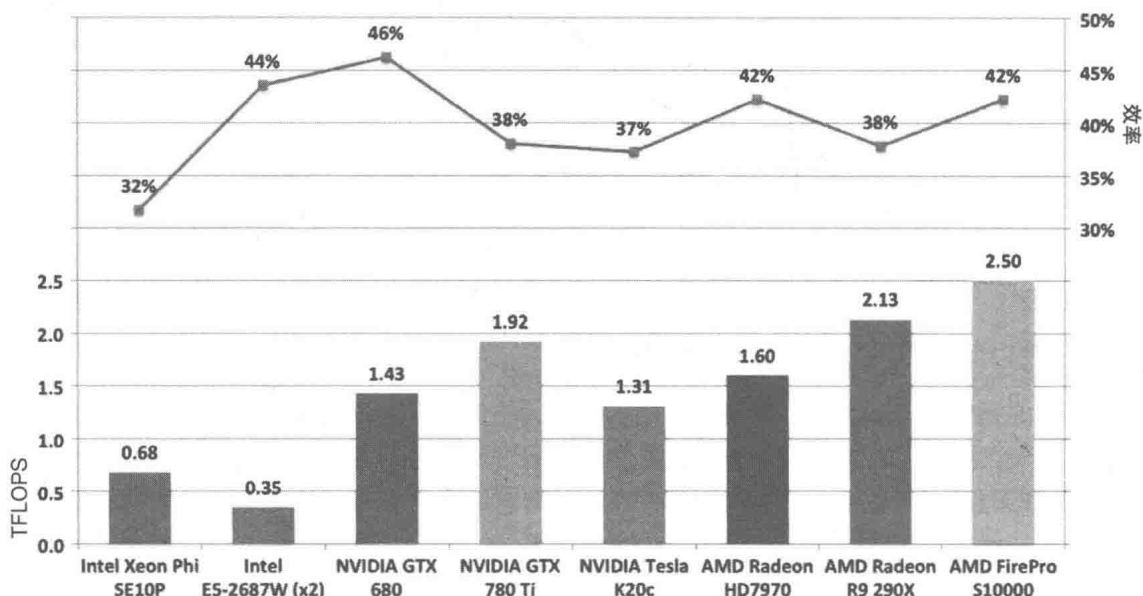


图 22-13 同一份源代码在不同众核和多核设备上的性能。图中性能为整个程序的运行性能

22.8 相关工作

OpenCL 和 NVIDIA CUDA 的核心思想是相同的。它们都适用于面向吞吐量的数据并行算法：其基本结构是在 1、2 或者 3 维的索引空间（核并行设计模式的实例）的每个点上都运行一个核实例。虽然两个编程系统的术语不同，但是理念相同。CUDA 和 OpenCL 的最大区别是：CUDA 只能用于 NVIDIA GPU，而 OpenCL 是一个被所有主流处理器厂商支持的可移植的开放标准。

对于喜欢进行处理器底层编程并能够管理并行平台细节的程序员来说，核并行是一个非常好的方法。然而，对于许多应用级程序员来说，这个方法过于底层。OpenMP 体系结构审核委员会已经关注到了这个问题，并在 OpenMP 4.0 版本中添加了针对不同设备的一组指令。这很好地整合了指令制导（OpenMP 采用的方法）编程的简单性和核并行模式（OpenCL 采用的方法）的高效性。

除了 OpenCL 行业标准、NVIDIA 的 CUDA 之外，还有一个同 OpenMP 4.0 类似但针对单一厂商产品的编程模型：OpenACC。OpenACC 是一个针对异构计算平台、采用指令制导方式的编程模型。如果程序员想要开发在多个计算平台上可移植的应用程序，应该选择 OpenCL 而不是 CUDA。同样的道理，从长远看，OpenMP 4.0 行业标准是比 NVIDIA 的 OpenACC 更好的编程选择。

22.9 总结

OpenCL 是一个相对较新的编程模型，支持程序员针对异构计算平台编写可移植和高性能的并行应用程序。虽然对于使用传统 MPI+OpenMP 编程模型的应用级程序员来说，OpenCL 可能太过于底层。但是，当将计算机作为一个异构系统并同时支持包括处理器、协处理器、GPU 和 FPGA 在内的所有计算设备时，OpenCL 是一个非常好的选择。正如本章所描述的那样，使用 OpenCL 完全可以写出高性能并且可移植的应用程序。

22.10 更多信息

下面是推荐的与本章相关的阅读材料。

- Munshi, A., Gaster, B., Mattson, T.G., Fung, J., 2011. OpenCL Programming Guide.
- Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D., 2011. Heterogeneous Computing with OpenCL.
- *Hands on OpenCL* 教程和代码示例（由 Simon McIntosh-Smith 与 Tom Deakin 提供），输入来自 Ben Gaster 与 Tim Mattson, <http://handsonopencl.github.io/>。
- McIntosh-Smith, S., Price, J., Sessions, R.B., Ibarra, A.A., 2014. BUDE paper: high performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications*. doi:10.1177/1094342014528252.
- McIntosh-Smith, S.N., Boulton, M., Curran, D., Price, J.R., 2014. On the performance portability of structured grid codes on many-core computer architectures. *International Supercomputing, Leipzig*. doi:10.1007/978-3-319-07518-1_4.
- Intel Xeon Phi 的 OpenCL SDK <http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor>.
- 本章和其他章的代码下载地址 <http://lotsofcores.com>。

应用到 Stencil 计算中的特性提取和优化方法

Cedric Andreolli*, Philippe Thierry*, Leonardo Borges†, Gregg Skinner†, Chuck Yount†

* 法国, Intel 公司; † 美国, Intel 公司

23.1 引言

本章描述了一个应用于 3D 有限差分 (3DFD) 算法的特征提取和优化方法。3DFD 算法常用于求解各向同性声波方程 (Iso3DFD), 而这种方程可以不限制密度是否为常数。

时间域有限差分是用于仿真地震波传播的一种常用技术, 仿真地震波传播进一步用于波现象分析和地震波获取设计。这个方法在地震成像中通过反转时间迁移和全波形反演而广泛应用。这个方法的变量包括将波看作声波或者弹性波, 以及在地表的物理描述中加入弹性变量和各向异性。

除此之外, 被选作近似偏导的 stencil 方法对于性能的实现来说有着巨大的影响。这些选择都会对 3DFD 算法的计算密度 (AI) 有影响 (计算密度是指每字节内存传输而带来的浮点运算次数)。使用 Roofline 模型方法时可以很容易地联系起 AI 和性能期望。这个方法有助于测量在特定硬件系统实现中给定程序的性能。Roofline 模型能够通过源代码的优化而设置期望的性能增益。一旦程序达到 Roofline 模型预测的性能等级, 就只能通过算法改进提升性能。

对于一台给定的计算机, 硬件规格决定了浮点计算能力 (FLOPS) 和从内存中传递数据和向内存传递数据的能力 (内存带宽)。我们可以通过运行标准的基准测试 (比如高性能 LINPACK 和 Stream Triad 基准测试) 来测定这些能力。

从最基础的 3DFD 实现开始, 我们用一个方法来评估这个算法在算法和硬件特征上能够达到的最好性能。

为了得到接近于期望值的性能, 本章描述了从最基本的版本到使用硬件内置函数的实现的一系列调试步骤。这个调试方法包括可扩展并行 (合作线程堵塞)、最大化内存带宽 (缓存分块、寄存器重用) 和最大化核内性能 (向量化、循环重建)。

我们接着介绍了一种可以求最优参数集的自动调试方法, 这些参数可能会在程序构建和运行时起作用。选择一组最优的优化参数是一项很难的任务, 因为复杂计算机系统中有许多不同的特征。调试参数源自手动调试方法、域分解影响、编译器能力和硬件影响等。这些参数通常源自源代码更改 (比如循环块的值)、编译器驱动的选项 (比如循环展开选项) 和硬件特征 (比如, 缓存大小)。

通常一个程序中都有十多个调试参数, 因为参数数量太多了, 所以简单的试错法根本没法起作用。自动调优是一种高雅的方法, 它在编译时和运行时优化代码以进行自动调优。它的主要想法是优化运行的输入参数和编译器标记, 而不会自动改变源代码。我们想要为指定应用找到一个已经优化的参数集, 而无论源代码是否经过优化。

我们使用遗传算法 (GA) 寻找可用参数空间, 包括缓存分块的大小、域分解形状、预

取参数和能量消耗。从结果上看,自动优化方法会比传统的费力的搜索方法要快得多。

从一个未优化的程序版本到一个高度优化的版本,我们在 Intel Xeon E5-2697 v2 处理器上得到了 6 倍的性能提升,在 Intel Xeon Phi 协处理器上得到近 30 倍的性能提升。除了性能提升之外,自动化调试方法为任何输入工作负载选择一个最优的参数集。

23.2 性能评估

Iso3DFD 内核可求解一个各向同性声波方程,这个方程在空间上有 16 阶,在时间上有 2 阶。Iso 3DFD 内核的一个标准实现通常只会达到硬件系统最高浮点运算能力 (FLOPS) 的 10%。本章将介绍如何在处理器和协处理器上得到 Iso 3DFD 内核的 Roofline 模型。为了计算出在给定平台上应用可达到的性能,需要如下性能参数:

- 计算的理论峰值:对于 Xeon Phi 7120A 协处理器,2420GFLOPS 的单精度运算速度和 352GB/s 的传输速度;对于两个带 1866MHz DDR3 内存的 Intel Xeon E5-2697v2 处理器,1036GFLOPS 的单精度运算速度和 119GB/s 的传输速度。
- 使用 Linpack (或者 DGEMM) 和 Stream Triad 基准测试达到的性能数值指定了这个平台性能的上限:对于 Intel Xeon Phi 7120A 协处理器,2178GFLOPS 的单精度运算速度和 200GB/s 的传输速度;对于两个 Intel Xeon E5-2697 v2 处理器,930GFLOPS 的单精度运算速度和 100GB/s 的传输速度。
- 应用的 AI 是根据浮点加法 (ADD) 与乘法 (MUL) 的次数和在内存上数据迁移的字节数 (用 LOAD 和 STORE 操作数量定义) 计算出的值。

最后有一个严格的假设:硬件有无限带宽缓存和零访问延迟的大小。这意味着一个完美的内存子系统,也就是说,一旦数据载入缓存中,它就会一直驻留在那里,而不影响程序运行速度。

当然,其他因素可能会影响到一个使用这类 3DFD 内核的应用的性能,这些因素包括:边界条件的选择、时间逆转的 I/O 方案和并行程序模型。对于本章来说,我们没有考虑边界条件或者 I/O。另外,在使用 OpenMP API 的硬件节点中,对于使用 MPI 标准以及共享内存线程的分布式内存并行性,并行实现的内核可能会用到异构的并行模型结合域分解。在本章中,我们只考虑在一个 SMP 节点处理一个域的情况。

23.2.1 测试平台的 AI

我们的测试系统由两个 Intel Xeon E5-2697 v2 (2S-E5) 组成,其中每个 CPU 中有 12 个内核,CPU 不使用 turbo 模式,两个 CPU 都在 2.7GHz 的频率下工作。每个处理器都支持 AVX 扩展,它使用 256 位的 SIMD 指令,这个指令能够在每个 CPU 周期中处理 8 个单精度 (32 位) 数字。因此理论的峰值是 $2.7\text{GHz} \times 8(\text{SP FP}) \times 2(\text{ADD/MULL}) \times 12(\text{内核}) \times 2(\text{CPU}) = 1036.8\text{GFLOPS}$ 。理论的内存带宽是从内存频率 (1866GHz)、通道数量 (4)、每个周期每个通道转移的字节数 (8) 计算来的因此这个系统有 $1866 \times 4 \times 8 \times 2(\text{处理器数量}) = 119\text{GB/s}$ 的峰值带宽 (对于双插槽 2S-E5 系统来说)。

我们也需要测量平台实际可以达到的值,以展示应用的行为。作为第一个近似,让我们考虑现实世界的应用性能总是由总内存带宽 (使用以 Stream Triad 为代表) 和总计算能力或者 FLOPS 表征 (以 Linpack 为代表)。两个基准测试的选择提供了一个非常严格的假设,同

时我们可能了解到：即便使用它们得到的数值离参考的理想点非常远，但比起硬件理论的峰值，这种峰值参数也更加可信，因为它们执行指令流的时候包含了处理设备所需要的最小开销。

在 2D-E5 系统上，Linpack 基准测试给出了 930GFLOPS 的峰值参数，Stream Triad 给出了 100 GB/s 的峰值参数。理论上 (AI_{cpu}^{th}) 和可达到峰值 (AI_{cpu}^{ac}) 分别为：

$$AI_{cpu}^{th} = \frac{1036.8}{119} = 8.7 \text{ Flop / Byte}$$

$$AI_{cpu}^{ac} = \frac{930}{100} = 9.3 \text{ Flop / Byte}$$

使用上述数字，我们能够刻画出一个给定内核的情况：如果内核的 AI 大于（低于）9.3Flop/Byte，我们可以定义它是计算限制（或者内存限制）的应用。在 Intel Xeon Phi 7120A 协处理器上，Linpack 和 Stream Triad 有 2178GFLOPS 和 200GB/s 的性能峰值，而它的理论峰值为 2420GFLOPS 和 352GB/s，因此，AI 是

$$AI_{phi}^{th} = \frac{2420.5}{352} = 6.87 \text{ Flop / Byte}$$

$$AI_{phi}^{ac} = \frac{2178}{200} = 10.89 \text{ Flop / Byte}$$

23.2.2 内核的 AI

Roofline 模型也需要计算给定应用的 AI。AI 的计算要么用肉眼观察统计代码中有多少次内存访问和计算，要么使用特制的工具来访问硬件计数器。在标准的 FD 内核中，我们发现 4 次加载 (c, prev, next, vel)，1 次存储 (next)，51 次加法（指数计算没有计入），以及 27 次乘法（见图 23-4）。AI 能够使用如下公式计算：

$$AI = \frac{\#ADD + \#MUL}{(\#LOAD + \#STORE) \times \text{word size}} \quad (23-1)$$

这个应用计算出 AI 是 3.9Flop/Byte，我们再用它乘以每个平台的带宽可以得到在协处理器上第一次估计的最大可达到性能为 1372.8GFLOPS，2S-E5 上是 464.1GFLOPS。然而，因为峰值 FLOPS 考虑了两个并列的管道（一个是 ADD，另一个是 MUL），所以一段没有完美平衡加法和乘法的代码很可能不会达到这样的峰值。因此，这个可达到的峰值应该用如下公式计算：

$$\frac{(\#ADD + \#MUL)}{2 \times \max(\text{add}, \text{mul})} \quad (23-2)$$

这表示能在 16FLOPS/cycle 中完成的总运算次数与 MUL 和 ADD 中最大值的比率，后者在假设只有一个 256 位 AVX SIMD 的情况下只能在 8FLOPS/cycle 中完成。它代表可以达到的峰值 FLOPS 的百分比。

图 23-1 和图 23-2 展现了 2S-E5 的上限是 354.9GFLOPS 和协处理器的上限是 1049.8GFLOPS 的 Roofline 模型，这都分别是从后一种 AI 计算方式中得到的。

一个更加贴近现实的 roofline 可以使用 Stream Triad 所得到的带宽乘以内核的带宽得到（分别为 390GFLOPS 和 780GFLOPS）。同时，当考虑到 ADD/MUL 不均衡（式（23-2））的时候，这种 roofline 会更加符合现状，如图 23-1 和图 23-2 中标注的理论带宽所示。

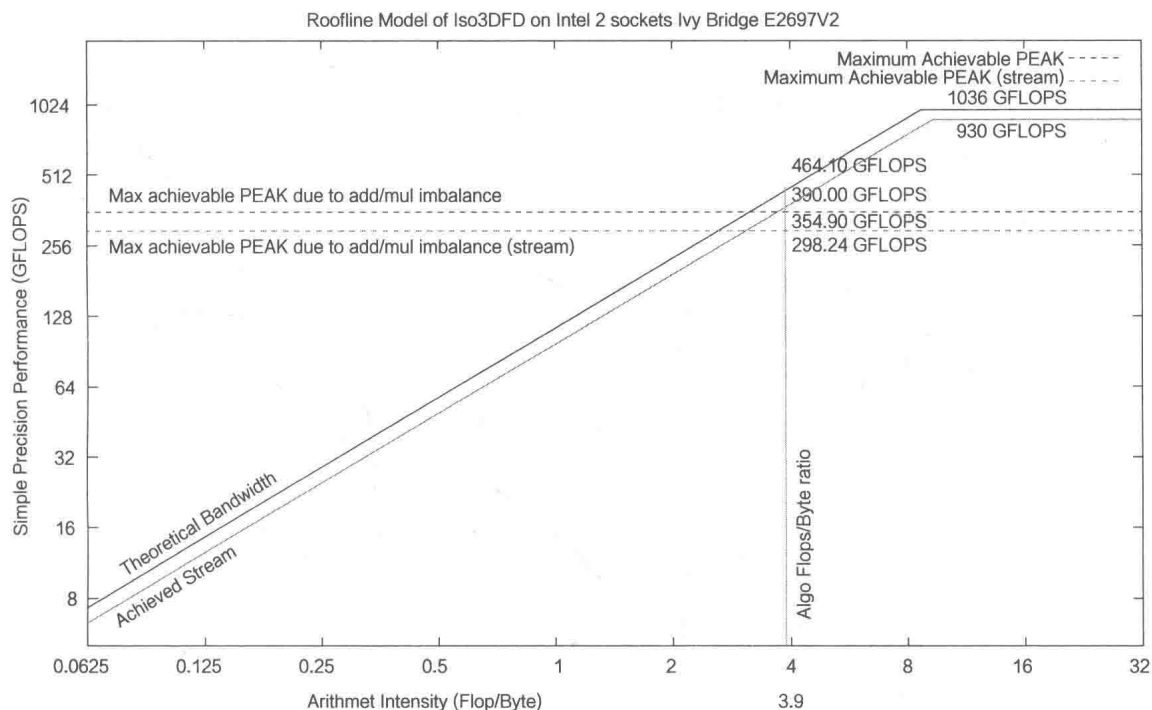


图 23-1 3DFD 在 Ivy Bridge 2S E5-2697 v2 上的 Roofline 模型。Roofline 分别代表了理论上限和平台可达到的极限。水平线代表当考虑到 ADD/MUL 不均衡和被 Stream Triad 带宽加权后的最大可达到峰值。垂直线代表 Iso3DFD 内核的 AI。和其他线的交叉点表明对应的可达到的极限

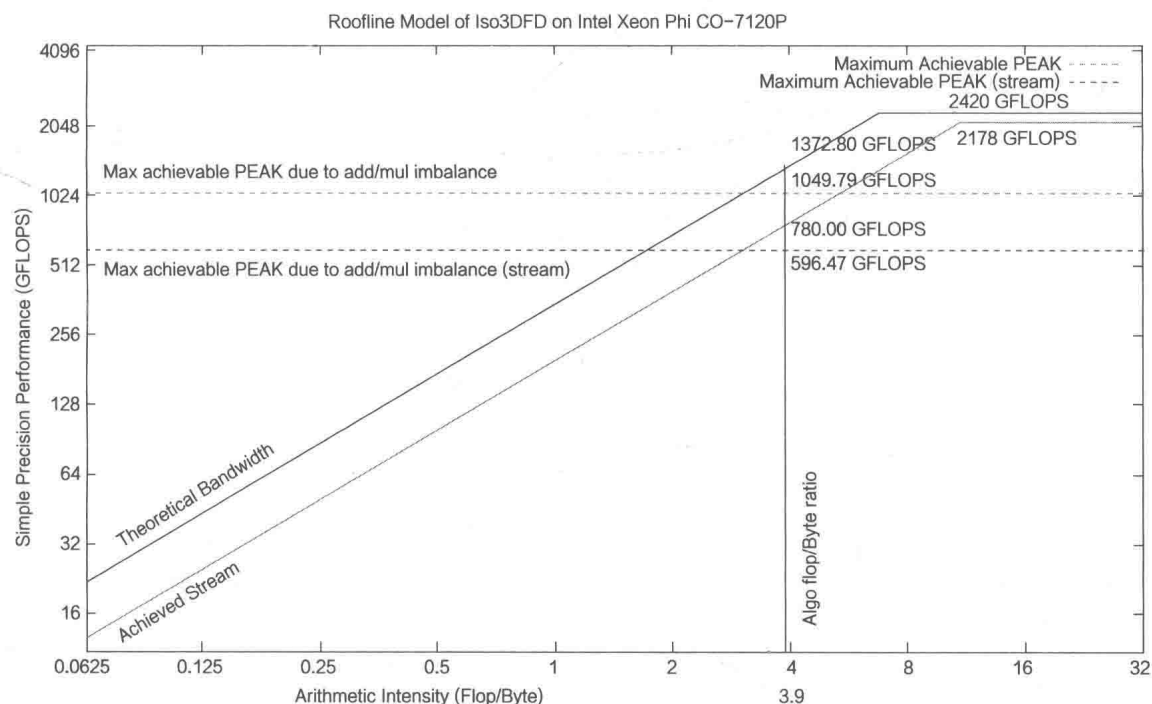


图 23-2 Xeon Phi 7120A 协处理器上的 Roofline 模型。Roofline 分别代表了理论上限和平台可达到的极限。水平线代表当考虑到 ADD/MUL 不均衡和被 Stream Triad 带宽加权后的最大可达到峰值。垂直线代表着 Iso3DFD 内核的 AI。和其他线的交叉点表明对应的可达到的极限

对于 2S-E5 来说,新的上限大约是 298GFLOPS,对于协处理器是 596GFLOPS。因为我们的模型是建立在完美缓存系统中的,所以这些数据依旧是高上限。如前面所描述,有人可能会通过加入一些比较低级的硬件特征得到更加现实的上限,比如缓存的影响和限制。仅有的缺失信息是为了得到这些上限而投入的实际的优化成本。

23.3 标准优化

标准优化的意思是为了提升并行化、向量化和数据局部性而做的修改。这三件事代表对于现在多核系统和多核架构优化最有冲击性的因素。我们通过如下步骤实现它们:

dev00: 标准的实现。纯粹的 3D 各向同性声波方程实现,它的结果是正确的(见图 23-3)。

```
for(int iz=0; iz<n3; iz++) {
for(int iy=0; iy<n2; iy++) {
for(int ix=0; ix<n1; ix++) {
    if((iz>=HALF_LENGTH && iz < n3-HALF_LENGTH) &&
        (iy>=HALF_LENGTH && iy < n2-HALF_LENGTH) &&
        (ix>=HALF_LENGTH && ix < n1-HALF_LENGTH)){
        int off = iz*n1n2 + iy*n1 + ix;
        float v = 0.0;
        v += prev[offset]*c[0];
        for(int ir=1; ir<=HALF_LENGTH; ir++) {
            v+=c[ir]*(prev[off+ir] + prev[off-ir]);
            v+=c[ir]*(prev[off+ir*n1] + prev[off-ir*n1]);
            v+=c[ir]*(prev[off+ir*n1n2] + prev[off-ir*n1n2]);
        }
        next[off] = 2.0f*prev[off]-next[off]+v*vel[off];
    }
}
```

图 23-3 未经过优化的内核源代码 (dev00)

dev01: dev00 在最内层循环中为了避免内存边界访问错误而加入的条件分支。在开始 AVX 指令集优化的时候,这些分支要使用掩码实现,移除分支并不影响 2S-E5 上的性能,但是会在协处理器上得到 2 倍的性能提升。事实上,多层循环要尽可能避免分支,以防编译器不能很好地处理掩码。编译器可能因为计算机的架构的复杂性(在 E5 中乱序执行引擎与在协处理器中顺序执行引擎)、指令集和向量长度(256 与 512bit)而不能完美地处理掩码。

dev02: 缓存分块会减少缓存命中的数量并且仅需要三个新的循环(见图 23-4)。这个方法的缺点是添加了新的参数去控制块大小。但是甚至不用优化这些参数,性能在协处理器上也能够得到巨大提升,当为了能够向量化而将倒数第二层循环(较快维度)的指数从 0 重新映射到 ixEnd 后,协处理器上的性能将会得到明显提升。

dev03: 为了保证线程中变量是线程私有的,并且这些私有变量不只在一次迭代中起作用,我们将 #pragma omp parallel for 语句分成 #pragma omp parallel 和 #pragma omp for 语句,并在两个 OMP 子句中插入了私有变量的声明。

dev04: #pragma ivdep 指令能够提示编译器去向量化数组,这些被向量化的数组在循环中一定要是没有别名的(假设别名是 C/C++ 编译器的默认行为)。向量化可以通过编译器参数(-fno-alias)或者使用 C/C++ pragmas 或者 Fortran 指令来启动。一个相对危险的方式是使用 #pragma simd 指令,这个指令强迫编译器向量化数组,但是如果代码中存在别名或者依赖,则可能导致不正确的结果。


```

for(int bz=HALF_LENGTHTH; bz<n3; bz+=n3_Tblock)
for(int by=HALF_LENGTHTH; by<n2; by+=n2_Tblock)
for(int bx=HALF_LENGTHTH; bx<n1; bx+=n1_Tblock) {
    int izEnd = MIN(bz+n3_Tblock, n3);
    int iyEnd = MIN(by+n2_Tblock, n2);
    int ixEnd = MIN(n1_Tblock, n1-bx);
    int ix;
    for(int iz=bz; iz<izEnd; iz++) {
        for(int iy=by; iy<iyEnd; iy++) {
            float* next = ptr_next_base + iz*n1n2 + iy*n1 + bx;
            float* prev = ptr_prev_base + iz*n1n2 + iy*n1 + bx;
            float* vel = ptr_vel_base + iz*n1n2 + iy*n1 + bx;
            for(int ix=0; ix<ixEnd; ix++) {
                float value = 0.0;
                value += prev[ix]*c[0];
                for(int ir=1; ir<=HALF_LENGTHTH; ir++) {
                    value += c[ir] * (prev[ix + ir] + prev[ix - ir]);
                    value += c[ir] * (prev[ix + ir*n1] + prev[ix - ir*n1]);
                    value += c[ir] * (prev[ix + ir*n1n2] + prev[ix - ir*n1n2]);
                }
                next[ix] = 2.0f* prev[ix] - next[ix] + value*vel[ix];
            }
        }
    }
}
}
}

```



```

__assume_aligned(ptr_next, CACHELINE_BYTES);
__assume_aligned(ptr_prev, CACHELINE_BYTES);
__assume_aligned(ptr_vel, CACHELINE_BYTES);
#pragma ivdep
for(int ix=0; ix<ixEnd; ix++) {
    v = prev[ix]*c0
    + c1 * FINITE_ADD(ix, 1)
    + c1 * FINITE_ADD(ix, vertical_1)
    + c1 * FINITE_ADD(ix, front_1)
    + c2 * FINITE_ADD(ix, 2)
    + c2 * FINITE_ADD(ix, vertical_2)
    + c2 * FINITE_ADD(ix, front_2)
    + c3 * FINITE_ADD(ix, 3)
    + c3 * FINITE_ADD(ix, vertical_3)
    + c3 * FINITE_ADD(ix, front_3)
    + c4 * FINITE_ADD(ix, 4)
    + c4 * FINITE_ADD(ix, vertical_4)
    + c4 * FINITE_ADD(ix, front_4)
    + c5 * FINITE_ADD(ix, 5)
    + c5 * FINITE_ADD(ix, vertical_5)
    + c5 * FINITE_ADD(ix, front_5)
    + c6 * FINITE_ADD(ix, 6)
    + c6 * FINITE_ADD(ix, vertical_6)
    + c6 * FINITE_ADD(ix, front_6)
    + c7 * FINITE_ADD(ix, 7)
    + c7 * FINITE_ADD(ix, vertical_7)
    + c7 * FINITE_ADD(ix, front_7)
    + c8 * FINITE_ADD(ix, 8)
    + c8 * FINITE_ADD(ix, vertical_8)
    + c8 * FINITE_ADD(ix, front_8)

    next[ix] = 2.0f* prev[ix] -    next[ix] + v*vel[ix];
}

```

图 23-5 内核 dev04 和 dev05 的源代码。这里 FINITE_ADD 是类型 $v[ix+off]+v[ix-off]$ 对称 FD 的宏

```

__assume_aligned(ptr_next, CACHELINE_BYTES);
__assume_aligned(ptr_prev, CACHELINE_BYTES);
__assume_aligned(ptr_vel, CACHELINE_BYTES);
#pragma ivdep
for(int ix=0; ix<ixEnd; ix++) {
    v = prev[ix]*c0
    + c1 * ( FINITE_ADD(ix, 1)
             + FINITE_ADD(ix, vertical_1)
             + FINITE_ADD(ix, front_1))
    + c2 * ( FINITE_ADD(ix, 2)
             + FINITE_ADD(ix, vertical_2)
             + FINITE_ADD(ix, front_2))
    + c3 * ( FINITE_ADD(ix, 3)
             + FINITE_ADD(ix, vertical_3)
             + FINITE_ADD(ix, front_3))
    + .....
    + .....

    next[ix] = 2.0f* prev[ix] -    next[ix] + v*vel[ix];
}

```

图 23-6 dev06 中的部分内核


```
#pragma ivdep
for (TYPE_INTEGER ix=0; ix<ixEnd; ix+=SIMD_STEP){
    SHIFT_MULT_INIT
    SHIFT_MULT_INTR(1)
    SHIFT_MULT_INTR(2)
    SHIFT_MULT_INTR(3)
    SHIFT_MULT_INTR(4)
    SHIFT_MULT_INTR(5)
    SHIFT_MULT_INTR(6)
    SHIFT_MULT_INTR(7)
    SHIFT_MULT_INTR(8)

    MUL_COEFF_INTR(vertical_1, front_1, coeffVec[1])
    MUL_COEFF_INTR(vertical_2, front_2, coeffVec[2])
    MUL_COEFF_INTR(vertical_3, front_3, coeffVec[3])
    MUL_COEFF_INTR(vertical_4, front_4, coeffVec[4])
    MUL_COEFF_INTR(vertical_5, front_5, coeffVec[5])
    MUL_COEFF_INTR(vertical_6, front_6, coeffVec[6])
    MUL_COEFF_INTR(vertical_7, front_7, coeffVec[7])
    MUL_COEFF_INTR(vertical_8, front_8, coeffVec[8])

    REFRESH_NEXT_INTR
}
```

图 23-7 在 dev08 中使用内置函数实现的宏

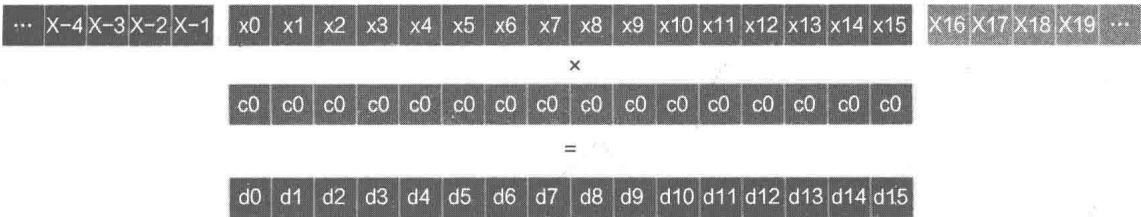


图 23-8 在协处理上的快速维度 (coefficient c0)

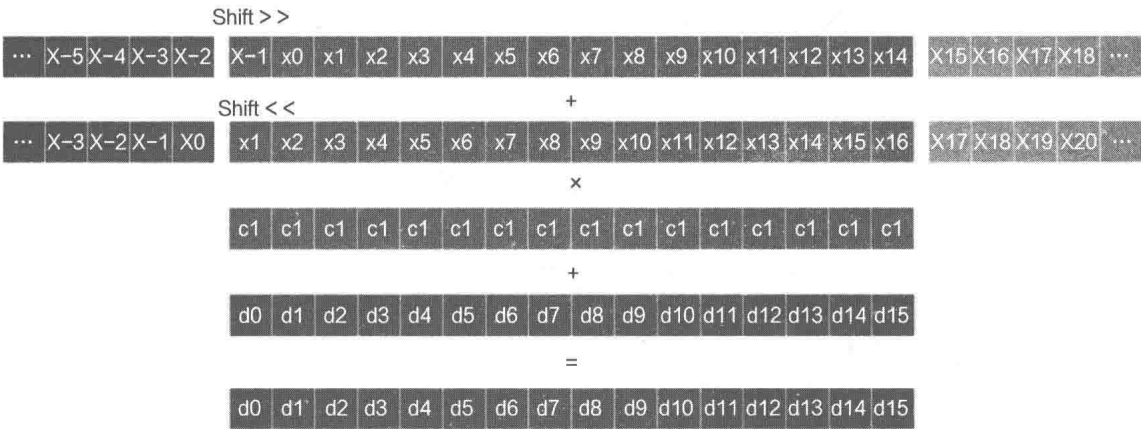


图 23-9 协处理上的快速维度向量化 (*coefficient c1)

对于其他两个维度，向量化是比较简单的。对于单个系数，只需要 4 次加载，接着这些向量加起来并乘以系数 (图 23-10)。这是在宏 MUL_COEFF_INTR (图 23-12 和图 23-13) 中实现的。

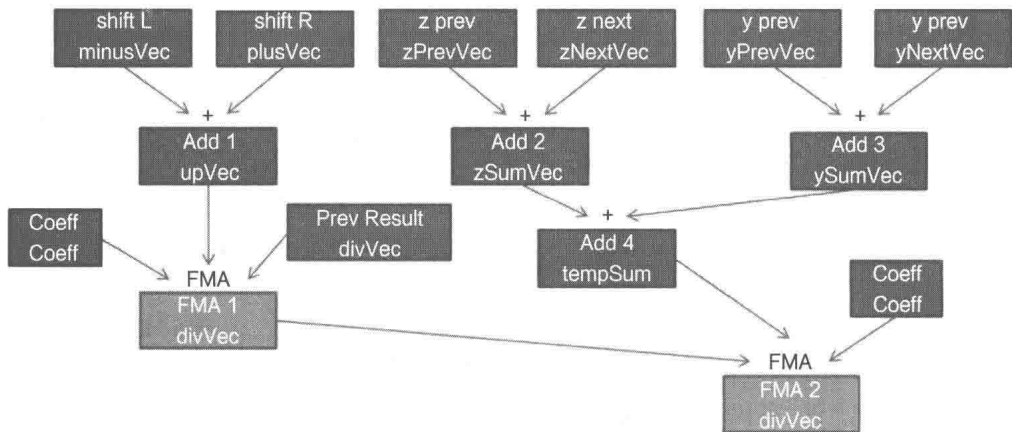


图 23-10 dev08 上对于单个系数的操作

dev09：在协处理器上，减少了临时变量的数量，同时，使用 FMA 指令（融合了乘加操作）也可减少了寄存器压力。系数在计算（6 FMA）的过程中能够保存在相同的寄存器中，并且每个 FMA 结果直接发送到下一组计算中，而数据只在寄存器间传输（见图 23-11）。在 AVX2 上 FMA 使用情况的讨论仍然在继续。

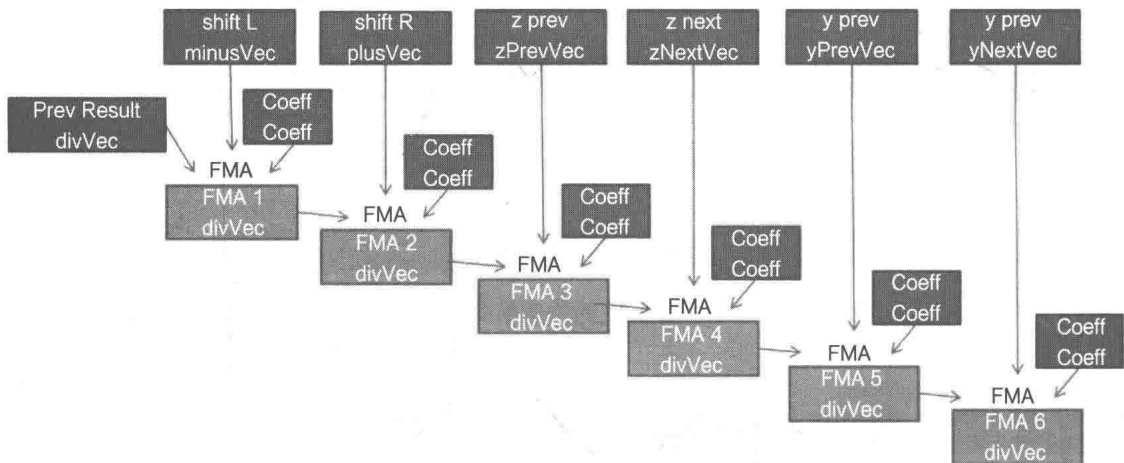


图 23-11 dev09 上对于单个系数的操作

	Ivy Bridge		协处理器	
	GFLOPS	占峰值性能的百分比	GFLOPS	占峰值性能的百分比
dev00	36.73	3.95	12.32	0.57
dev01	35.97	3.87	22.36	1.03
dev02	42.49	4.57	141.88	6.51
dev03	42.58	4.58	147.90	6.79
dev04	45.16	4.86	148.14	6.80
dev05	90.45	9.73	156.14	6.45
dev06	91.58	9.85	182.39	8.37
dev07	137.96	14.83	176.09	8.08
dev08	125.40	13.48	208.46	9.57
dev09	132.48	14.25	208.97	9.59
dev09+GA	226.27	24.33	368.46	16.92

图 23-12 以 GFLOPS 为单位表示协处理器上 ECC off/Turbo 和 Ivy Bridge 上 Turbo 的性能

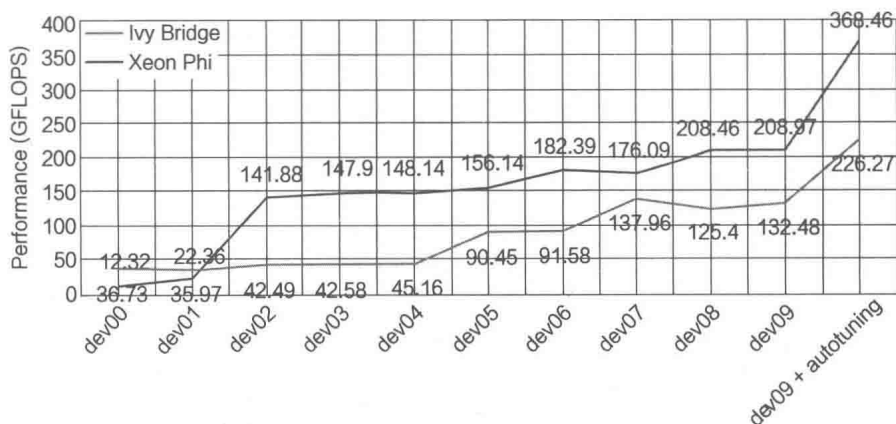


图 23-13 在 2S-E5 和协处理器上，每个版本的效率。最优版 dev09 也同时被遗传算法自动调整优化过

23.3.1 自动应用调试

因为微架构变得越来越复杂，以及数量不断增加的调试标志和编译指示或者指令，所以编译器也更难以使用。在多维参数空间中的优化问题尽管可以手动处理，但是通过数学优化会更容易一些。

为了得到优化参数，基于上述想法，我们开发了一个基于 GA 优化的自动调试框架。我们的工具就像编译器一样运行，并使用 GA 算法求出一个效率高的参数集。

为这样的优化问题而选择正确的方法并不是显而易见的，23.5 节中的出版物，如“Introduction to Stochastic Search and Optimization” (James, 2013) 或者“Metaheuristics—The Metaphor Exposed” (Sörensen, 2013)，可能可以帮助你更好地理解启发式算法研究领域和随机优化的特点。同时我们已经提供了一个链接，其中含有 GA 的开源库实现，以便创建相似的框架。

我们的目标是选择一个优化算法，即便在不完整、不完美的信息中也能得到好的解。就像任何非耗尽优化算法或者迭代算法（比如模拟退火算法）一样，并不能保证得到的是全局最优解。

很多的启发式算法用随机算法的一些形式实现，这样解的产生依赖于产生的随机变量集。GA 以离散和结合优化问题为目标，比如 3DFD 案例。

GA 操作总体。一个 GA 选择一代中几个最优的个体并产生新一代，新一代包括比前一代更优的特性。在本例中，第一代是随机产生的，并用来产生第二代。这个过程可以用来重复运算很多次。一个群体由不同个体组成，一个个体是由多个编译器优化选项、一个展开因素、执行域、缓存块大小、MPI 域的形状和大小以及能量消耗等属性描述的。

我们认为输入域的尺寸（速度、密度、各向异性参数）是固定的，因为它们是从先前处理步骤中得到的。当在每个节点分配一个或者多个震源 gather 的时候，这些输入大小不能更改。然而，当考虑到一个域分解实现时，我们能够在每个共享内存节点优化域大小。为了创造新一代，GA 与 4 个主要概念相关（图 23-14）。

评估：对于每个个体，提供一个评估标准。这个评估标准叫作适应性（fitness）。我们想减少流逝的时间（低适应性的评估）或者增加 GFLOPS（高适应性的评估）来作为评估标准。

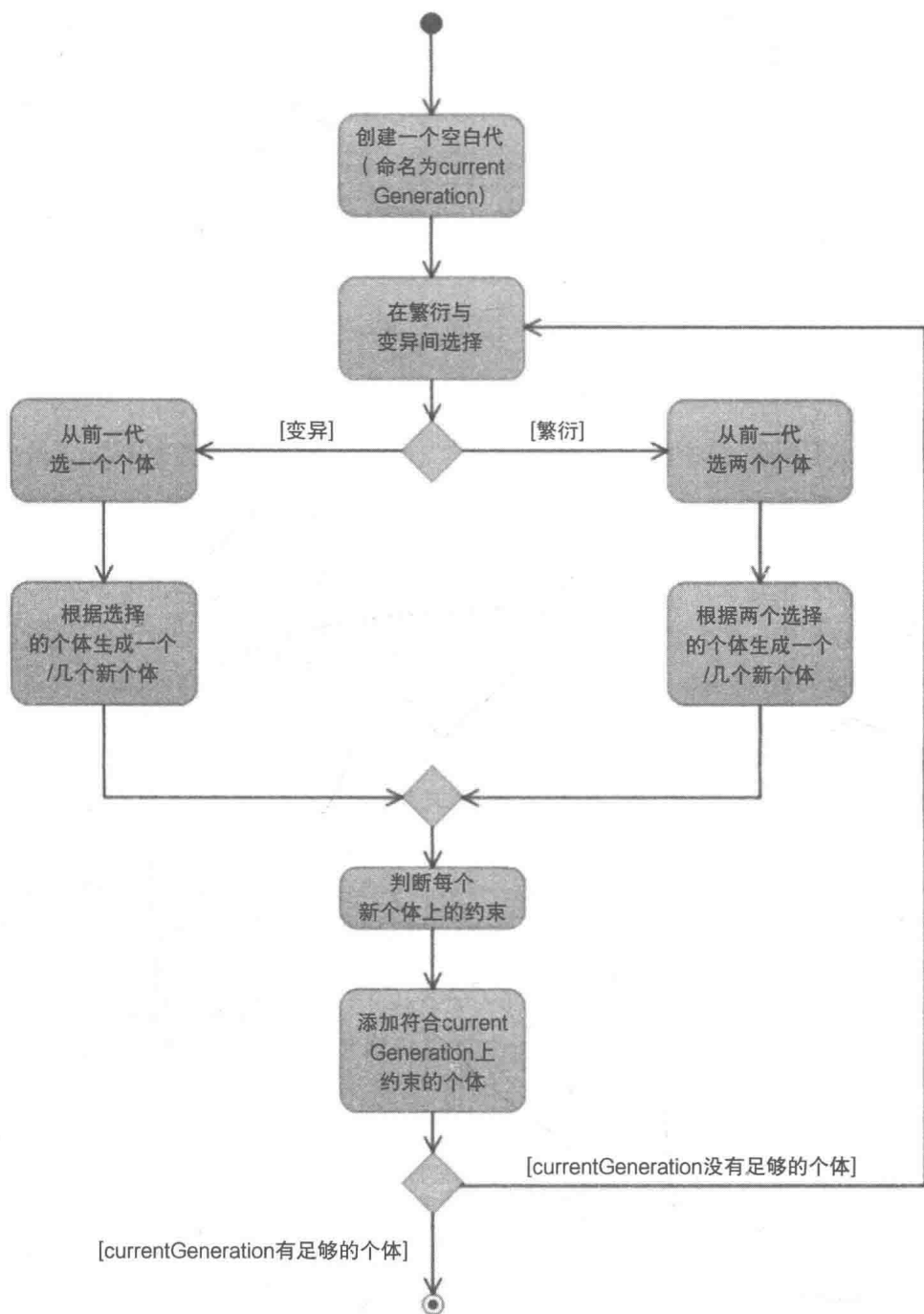


图 23-14 遗传算法优化的一般流程图。在本例上，我们将流逝的时间作为评估标准

挑选：挑选允许我们在现存的个体中挑选一些个体去繁衍和变异。挑选必须支持最好的个体，但是也必须选择一些“坏”个体，这是为了避免过快收敛到局部最优解。多数时候，选择这步操作是基于适中的非一致分布的随机选择。轮盘法可以用来挑选个体（图 23-15）。因此一个个体若有更好的适应性，被选择的机会则会更高。第一步是评估每个个体的适应性，接着根据适应性排序这些个体。在最后一步中，先选择一个介于 0 到个体总数的随机数。这群个体被叠加起来并且选择出适应性匹配的个体。

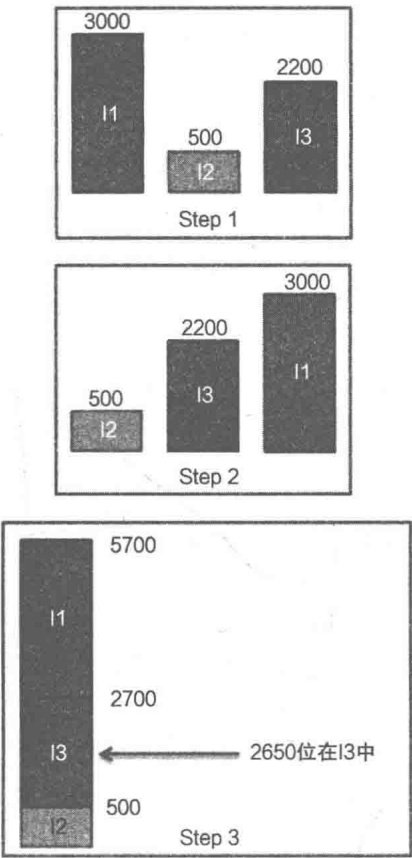


图 23-15 轮盘法的描述，这种方法中个体根据其适应性分布。这个选择支持：最优个体一定会被选择，一些“坏”的个体也有可能被选择，这样可以避免过快收敛到局部最有优解。在这个案例中，步骤 3 中，2650 是一个随机数，它选择了个体 I3

繁衍：繁衍是一个允许 GA 朝着最优解方向收敛的过程。前一代中的两个或者多个个体融合从而创造出来一个或者多个新的个体。给定两个父辈，轴代表用来分裂和结合成为新个体的索引。首先随机选择轴的数字，接着随机选择每条轴的索引（图 23-16）。

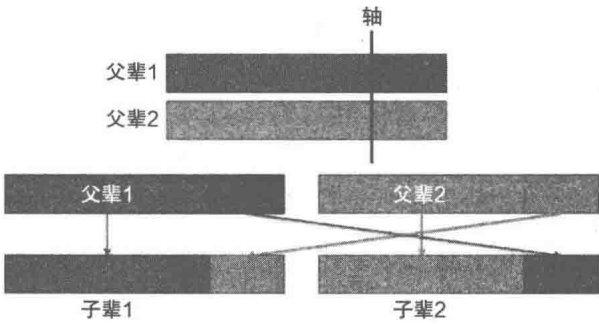


图 23-16 沿一条轴繁衍的解释。繁衍是一个允许 GA 朝着最优解方向收敛的过程。给定两个父辈，轴代表用来分裂和结合为新个体的索引。首先随机选择轴的数字，接着选择每条轴的索引

变异：变异保证了对参数空间的探索并避免过快收敛到局部最优解。因此我们随机选择属性进行修改并赋予它们随机值（图 23-17）。

在描述 GA 的上述四步中，计算密集度最高的步骤是评估步骤，这一步中，为每次实验建立并运行应用程序。这点对于任何搜索算法来说都是关键点，因为评估给定参数集所需要的时间将会对优化方法的选择有影响。如图 23-18 所示，我们可能需要上千次实验，并且，如果测试用例的时间不够短甚至 GA 在并行模式下都运行，完成这个测试是不可能的。

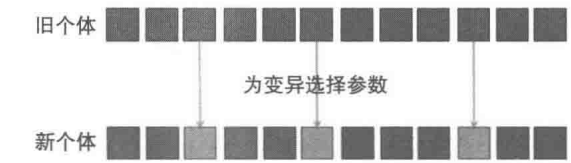


图 23-17 变异的解释。变异保证了参数空间的探索并避免参数过快朝着局部最优解的方向收敛

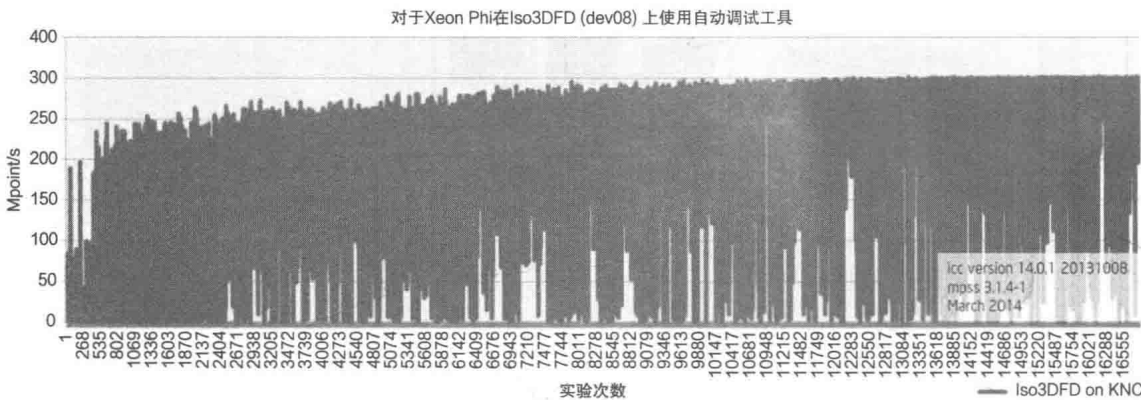


图 23-18 在协处理器上用自动调试工具优化 Iso3DFD 参数

对于时间步应用（这种应用中，一些（密集型的）计算是通过每次迭代实现的），我们接着能减少时间步的数量去评估参数调整的冲击。初始化会影响最少迭代数目的选择，这样才能保证没有缓存效应影响搜索。

做优化或做基准测试时所处的情况是相同的。为了保证任何修改所需要的评估时间都不会超过一分钟，并且保持应用具有代表性的特征，选择合适的测试用例是极度重要的事情。在 MPI 域分解形状影响不同节点通信的情况下，测量单节点足以对一些主要的参数（缓存块、编译器参数等）进行调优。当然，为了优化域分解形状的影响或者调试 MPI 环境变量，相同的方法能够应用到集群级别的优化上，但关键依旧是保证一个节点运行过程的代表性。另一个缩短收敛时间的方法是在多个计算系统中并行估计多个个体。

23.3.2 自动调试工具

我们为优化程序的构架和执行开始了一个工具。我们的自动调试工具可以进行如下配置。

- 描述项目（路径、工作目录、代数、个体数等）。
- 描述一个“节点”集合。对于每个节点，线程在机器上创建，它运行自动调试工具。这个线程启动编译并使得运行程序以并行优化多个节点。
- 定义变量（int、float、stringset）和它们可以使用的值，这些值将用于配置文件的其他地方；比如：

```
int v1[1, 100, 5]
float v2[0.1, 12.6]
stringset v3["This", "is", "an", "example"]
```


- 加入背景知识，这些背景知识要在先验 (a priori) 信息或者限制形式下出现。一个个体能够加入当前一代只有它满足所有限制。比如：这里有一个关于 $n1$ 、 $n2$ 、 $n3$ 的限制： $n1*n2*n3 < 100,000$;

```
begin constraints
  n1*n2*n3 < 100000
end constraints
```

- 配置编译和执行的环境
- 建立并结合编译或者执行调用的命令：

```
begin buildcall
  "make clean; make"
end buildcall
begin programcall
  "./run_on_mic.pl"; "bin/iso3dfd.exe"; NODE; n1; n2; n3; cb1; cb2; cb3; nbThreads
end programcall
```

- 指明一个正则表达式从程序输出中获取适应性。在本例中，Iso3DFD 输出速度是 GFLOPS 为单位的。正则表达式语法使用 Perl。接下来的样例展示了如何匹配 Iso3DFD 的数值结果。

```
begin regex
  "speed:\s*([0-9\.]+)\s*GFlop\/s"
end regex
```

23.3.3 结果

图 23-13 展示了在 $n1 = 256$ 、 $n2 = 200$ 和 $n3 = 400$ 数据集情形下的性能结果，这个优化修改了 10 个版本 (dev 00 到 dev 09)。这里，在 2S-E5 (24 个线程) 和协处理器 (244 个线程) 上达到了大致又 6 倍和 30 倍的性能提升，自动调试工具仅仅应用到 dev09 版本上，相比于原始的版本，它提供了两倍的性能提升。图 23-18 展示了为自动调试工具做每次尝试之后以 GFLOPS 为单位的性能提升情况 (越高越好)。整体性能逐步增加到超过 350GFLOPS 图 23-19 展示了通过自动调试工具产生的参数，它为 dev02、dev07 和 dev09+GA 提供了更高的性能。

对于 2SE5 和协处理器，我们分别达到了最高性能的大约 70% 和 60%。考虑到 roofline 模型中的完美缓存假设，这一句是非常好的结论。效率在协处理上稍微差一点是出于顺序微处理器和大量线程数的缘故。典型地，任何优化都会影响两个平台，但是我们注意到在协处理器上最大的影响因素是：

- 缓存块和向量化
- 内核分解指令减少
- 内置函数

但是在 2S-E5 平台上最大的影响因素来源于：

- 手动展开和 `_assume_aligned` 指令
- 第一次接触

23.4 总结

除了讨论上述方法的性能提升之外，本章中最重要的方法是保证高性能的三步方法：

版本	平台	GA	循环顺序	相关性	优化	预取	nx	ny	nz	cbx	cby	cbz	线程数	性能 (GFLOPS)
Dev02	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	compact	"-O3"	Not used	400	400	400	400	40	40	24	42.49
	Xeon Phi 7120 C0	否	"-DBLOCK Y X Z"	compact	"-O3"	"-opt-prefetch-distance=84,48"	272	1105	1452	1392	12	199	24	48.16
Dev07	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	balanced	"-O2"	Not used	400	400	400	400	4	40	24	141.88
	Xeon Phi 7120 C0	否	"-DBLOCK Z X Y"	balanced	"-O3"	"-opt-prefetch-distance=4,2"	368	316	1403	1488	3	12	183	217.39
Dev09	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	scatter	"-O3"	Not used	256	256	256	256	2	40	24	137.96
	Xeon Phi 7120 C0	否	"-DBLOCK Z Y X"	compact	"-O3"	"-opt-prefetch-distance=88,14"	208	1227	1305	1216	49	70	24	217.38
Dev09	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	balanced	"-O3"	Not used	400	400	400	400	4	40	24	176.09
	Xeon Phi 7120 C0	否	"-DBLOCK X Y Z"	balanced	"-O3"	"-opt-prefetch-distance=4,2"	416	278	1427	1184	3	9	183	248.46
Dev09	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	分散	"-O3"	Not used	400	400	400	132	40	40	24	132.48
	Xeon Phi 7120 C0	否	"-DBLOCK X Y Z"	紧密	"-O3"	"-opt-prefetch-distance=78,24"	224	1222	1405	1376	48	93	24	226.27
Dev09	2 Sockets E2697 V2	否	"-DBLOCK Z Y X"	均衡	"-O3"	Not used	400	400	400	400	1	40	240	208.97
	Xeon Phi 7120 C0	否	"-DBLOCK X Z Y"	均衡	"-O3"	直接在内置函数中	400	388	1432	400	1	48	244	368.46

图 23-19 用于优化步骤 dev 02、dev 07 和 dev 09+GA 的 GA 自动调试工具产生的参数

- 在开始调试之前评估可到达的最高性能。
- 调试代码实现并行性、数据局部性和向量化。
- 为构建和运行时阶段自动调试两组最优参数。

为了提升性能而建立策略是一个复杂的任务，在制定这项策略前我们应该知道这个应用在给定平台上的功能。如果我们只关注总共流逝的时间，而对最优可达性能没有任何概念，是一点意义都没有的。计算 Roofline 模型是了解性能上限最直接的方法。一旦结束这项任务，大多数时间将花在手动优化阶段。现在已经有了很多很完善的策略，比如自顶向下迭代（参考样例：<https://software.intel.com/en-us/articles/de-mystifying-software-performance-optimization>）。

必须时刻记住，对于第一序近似，至少在节点级，大多数高性能科学应用都被 FLOPS 或者 Bytes/s 所限制，即所谓的 CPU 和带宽限制。当应用迁移到集群级后，关键因素被浓缩在了 Amdahl 法则中，关键因素也包含在实现负载均衡的需求中，从而弥补任何工作负载或硬件的不均衡。

深入到代码中，代码的向量化（或者 SIMD 化）对于达到最优性能来说是另一个关键因素。但是，利用最大的 SIMD 性能将依赖于平台的带宽能力和 FP。

“最佳优化”并行实现之后，我们相信自动调试将成为提取计算系统中最佳性能的关键一步。再一次，“最佳优化”可以根据其他标准而定义，不仅仅是简单的性能。当我们可以将其 Roofline 模型中定位应用程序时，我们就可以估计怎样进一步优化应用。如果我们参考本章中 10 个相同版本的 3DFD 内核，那么很清楚的是，开发和维护成本在原始 dev00 版本和内置函数 dev09 版本不同。

一旦选择基于行业或研发标准，使用一个自动调试工具来得到最优参数而无需修改源代码是一个非常优雅和简单的方法，这种方法可以让代码得到相当大的改进，同时保持代码的可移植性和可维护性。还要注意，没有必要为每个更改或每次代码的运行而运行 GA 自动调试工具，只有当硬件或工作负载特征显著变化时才有必要这么做。

23.5 更多信息

- Sörensen, K., 2013. Metaheuristics—the metaphor exposed. International Transactions in Operational Research. <http://dx.doi.org/10.1111/itor.12001>.
- Spall, J.C., 2003. Introduction to Stochastic Search and Optimization, first ed. John Wiley & Sons, Inc., New York, NY.
- Williams, S., Waterman, A., Patterson, D., 2009. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM. 52 (4), 65-76 <http://doi.acm.org/10.1145/1498765.1498785>.
- A large variety of genetic algorithm software can be found on <http://www.geneticprogramming.com/ga/GAsoftware.html> and <http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/systems/0.html>.
- Several genetic algorithm libraries for different programming languages can be found at <http://geneticalgorithms.ai-depot.com/Libraries.html>.
- Download all codes from this, and other chapters, <http://lotsofcores.com>.

剖析指导优化

Andrey Vladimirov

美国, Colfax International

剖析指导优化是一种使用应用程序的性能分析(剖析)工具提高开发人员工作效率的优化方法。基于剖析工具反馈的结果,程序员可以优化源代码以提高处理器及协处理器上程序的性能。众所周知,通用的剖析工具可以识别代码中的“热点”,即代码执行过程中耗费绝大部分时间的函数或者代码段。因此,使用剖析工具(如 Intel VTune Amplifier XE)可以进一步提示程序员关于限制程序性能的一般性问题,如缺乏并行性、负载不均衡及高延迟操作等。

本章演示的剖析指导优化基于 Intel 架构。示例代码中涉及存储器和缓存流量调优的难题,其关键在于方阵的就地转置问题。矩阵转置可以用来解决许多计算问题,包括离散快速傅里叶变换、线性代数中的数值方法、图像处理及计算流体动力学。由于矩阵转置不涉及算术运算,因此其性能主要取决于内存及缓存的访问带宽。实验证明在矩阵转置过程中,可以通过优化数据局部性、规范化内存访问模式及使用并行性有效提升内存访问带宽。应用剖析工具(Intel VTune Amplifier XE)可以有效指导 Intel Xeon 处理器和 Intel Xeon Phi 协处理器平台上算法优化策略的使用。本章的实验结果和实验方法适用于多核 Intel 处理器(CPU)平台以及基于 MIC 架构的众核协处理器平台。

24.1 计算机科学中的矩阵转置

本章剖析指导优化中测试用例的计算“热点”即为矩阵转置。二维矩阵转置是指重新排列矩阵元素,使矩阵的行改变成列,列对应改变成行。矩阵转置通常应用于多维数组的预处理,目的是使对多维数组的数据访问具有连续性,避免数据访问出现较大步长。例如在线性代数运算(如 xGEMM)中,较优的做法是首先进行矩阵转置,以达到数据连续访问的目的。矩阵转置是一些离散快速傅里叶变换方法的基础(见 24.9 节第 3 条文献)。同样计算流体动力学中也采用了多维数组的转置,例如模板解算器在多维数组中的每个维度都使用了转置运算(见 24.9 节第 6 条文献)。然而,若试图使用转置运算对其他算法进行预处理并获得加速效果,则转置运算本身必须是高效的。

本章的讨论仅限于共享存储器中方阵元素的就地转置。‘就地’意味着转置结果矩阵覆盖原始矩阵,且共享存储器转置算法中的数据存储在单一计算节点,而不是整个集群。本章内容仅限于方阵,长方形矩阵转置是一个更加复杂的问题,但是其同样可以从方阵加速计算中得到启发(见 24.9 节第 4 条文献)。

数学上,转置定义如下:

$$A_{ij} = A_{ji}, \quad i = 0, 1, \dots, (n-1), \quad j = 0, 1, \dots, (n-1)$$

在计算机存储器中,矩阵 A 中元素 A_{ij} 存储在距离矩阵起始地址偏移量是 $i \times n + j$ (行优先矩阵)或者偏移量是 $j \times n + i$ (列优先矩阵)的位置。为了不失一般性,本章分别讨论以上两种情况。

Intel Xeon 处理器和 Intel Xeon Phi 协处理器平台下，行优先矩阵转置的难题在于数据的移动以 64 个字节的数据块（缓存行）为单位。如果数据是双精度（每个数据 8 字节），则每个缓存行包含 8 个矩阵元素。因此，程序中即使每次只取 1 个矩阵元素，从内存中获取该元素的过程中缓存也取得了另外相邻的 7 个元素。如果缓存行中所有的 8 个元素依次参与计算，则该种方式能够取得很好的性能。然而，如果存储器访问模式是处理器一次从缓存行中取 1 个数据，则处理器指针移动是基本需求的 8 倍，这极大地影响程序的性能，但是这种情况在转置运算中十分常见。例如在行优先矩阵中交换多个元素 A_{ij} 为 A_{ji} ，如果数据访问是连续的（即坐标 j 是顺序增加的，如 A_{ij} , $A_{i(j+1)}$, $A_{i(j+2)}$ 等），那么取数据过程中每个缓存行中的所有数据都被命中。然而，写数据过程中对应位置 A_{ji} , $A_{(j+1)i}$, $A_{(j+2)i}$ 数据都命中在不同的缓存行中（假定数据是双精度，且 $n > 8$ ）。同样，该理论适用于列优先矩阵。数据访问示意图见图 24-1。

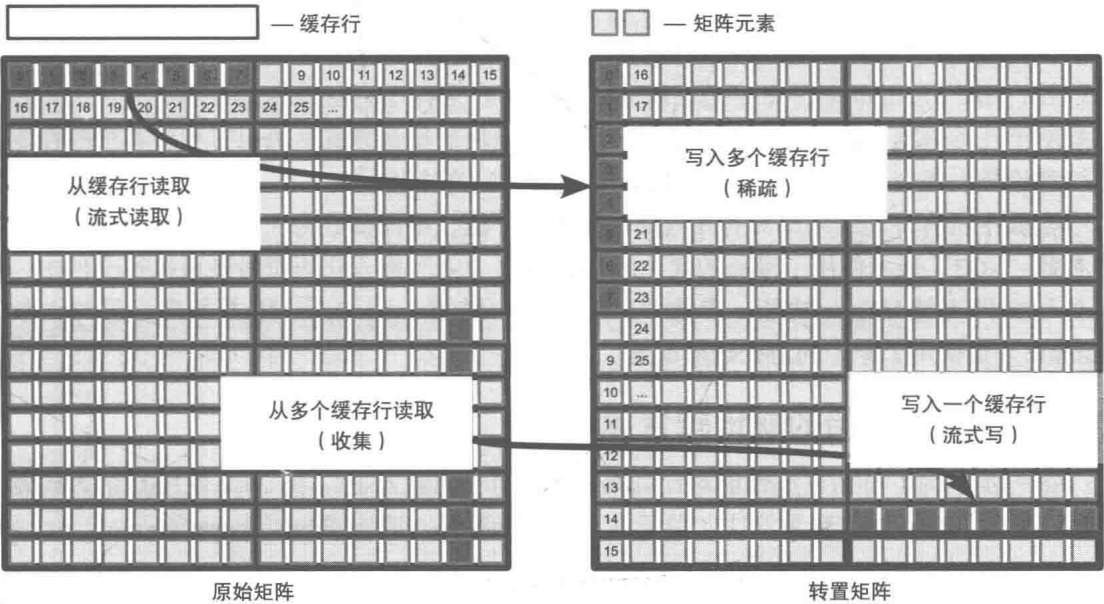


图 24-1 行优先矩阵转置。随着算法遍历数据，出现稀疏访问内存的情况，导致处理器访问多个缓存行，每个缓存行中的 8 个矩阵元素只有 1 个是有用的

当然，再一次使用一个缓存行时，缓存可以缩短数据访问延迟。例如，当交换矩阵元素 ($A_{i0} \leftrightarrow A_{0i}$) 时，缓存行中数据 $\{A_{0i}, A_{0(i+1)}, \dots, A_{0(i+7)}\}$ 被读入处理器缓存中。若顺序交换数据 ($A_{i1} \leftrightarrow A_{1i}$, $A_{i2} \leftrightarrow A_{2i}$ 等)，则缓存行中数据 $\{A_{0i}, A_{0(i+1)}, \dots, A_{0(i+7)}\}$ 没有重用。然而，在处理器交换数据 ($A_{i(n-1)} \leftrightarrow A_{(n-1)i}$) 并且开始交换 ($A_{(i+1)0} \leftrightarrow A_{0(i+1)}$) 时，缓存行中第 2 个元素得到了重用。如果缓存行中数据在下一次数据访问中命中，则数据访问时间会比直接从主存储器中读取数据更短。

自然地，为了使缓存行更加有效地工作，算法必须考虑缓存的特点。如果矩阵尺寸 n 足够大，算法顺序从 A_{i0} 到 $A_{i(n-1)}$ 遍历矩阵中的元素，则包含 A_{0i} 的缓存行将不会再放入处理器缓存中。但是，当算法交换元素 ($A_{(i+1)0} \leftrightarrow A_{0(i+1)}$) 时，包含元素 A_{0i} 的缓存行又会重新被命中。因此，更智能的做法是当数据还存在缓存中时就应该在下次数据访问时重用。

为了解决上面讨论的问题，在后续的章节中我们采用了性能剖析工具。

24.2 工具和方法

虽然本章目的是创造一种可移植、面向未来的代码，但是软件工具和微处理器的演化可

能会影响实验结果和实验方法的使用。为了能够重现实验结果，下面简单介绍实验中使用的系统配置。

下面的所有实验都在 Colfax SXP7450 工作站中完成（见 24.9 节）。Colfax SXP7450 拥有一块 Intel Xeon E5-2630v2 两路处理器（共有 12 个 2.6GHz 的物理内核），拥有 1 块容量为 128GB、频率为 1066MHz 的 DDR3 主存储器，拥有两块核数为 57、频率为 1.1GHz、全局内存为 6GB 的 Intel Xeon Phi 3120A 协处理器。平台系统为 CentOS Linux 6.5，系统内核为 kernel 2.6.32-431.el5.x86_64 和 MPSS 3.2.3，Intel C++ 编译器版本为 14.0.2.144(Build 20140120)，剖析工具版本为 Intel VTune Amplifier XE 2013 Update 15(Build 328102)。

实验中使用带宽作为衡量矩阵转置的性能指标，单位是 GB/s，带宽定义如下：

$$P = \frac{2 \times \text{size of (double)} \times n \times n}{10^9 \times T} \text{ GB/s}$$

其中， n 是矩阵大小；系数 2 指在矩阵转置过程中每个矩阵元素从主存储器中分别有读写两个过程；size of (double) = 8 字节；除以 10^9 表示单位从 B/s 到 GB/s 的系数； T 是矩阵转置过程所消耗的时间，单位为秒。带宽是一种非常通用的衡量方式，可以方便将算法的性能和系统理论最大集合（比如，通过 STREAM 基准测试）做对比。

实验中所有性能结果都具有相同的编译指令环境，编译参数为“-O3 -g -fopenmp”。其中参数“-O3”表示使用了最高的优化级别，参数“-g”嵌入代码符号到可执行文件中，使可执行文件支持 VTune 功能，参数“-fopenmp”链接 Intel OpenMP 并行库。另外，参数“-mmic”指使用了原生的协处理器配置。

24.3 串行：初始的就地转置实现

第一步，为了得到算法性能对比的基准（从简单问题入手），图 24-2 使用 C 语言构建方阵的“就地”串行矩阵转置算法。关于本章的算法源码，请访问 lotsofcores.com。

```
#define FTYPE double

void Transpose(FTYPE* const A, const int n) {
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < j; i++) {
            const FTYPE c = A[i*n + j];
            A[i*n + j] = A[j*n + i];
            A[j*n + i] = c;
        }
    }
}
```

图 24-2 方阵就地转置的串行实现

本章剩余部分中，我们采用的转置矩阵大小 $n = 4000$ 。因此在双精度 4000×4000 的矩阵中，矩阵大小为 122MB，远大于 CPU 中的 30MB 末级缓存，且大于 CPU 末级缓存外加协处理器的 28.5MB 二级缓存之和。如图 24-3 所示，基准测试过程中包含多个转置函数调用，这里前两个转置函数调用的时间不计算在内。

示例代码是 Intel Xeon 处理器和 Intel Xeon Phi 协处理器平台下本机模式下的基准测试。本机模式是指该代码编译参数是“-mmic”，我们需要将算法的可执行文件复制到协处理器系统下执行（见 24.9 节第 9 条文献）。

图 24-27 是该示例及其后续优化示例的平均性能示意图。图 24-2 中代码的运行性能在

图 24-27 中由“Serial”标签标记。在主机端，算法的带宽性能是 4.3GB/s；同时在协处理器端带宽性能为 0.7GB/s。

```
const int nTrials = atoi(argv[2]);
const int skipTrials = 2;
for (int iTrial = 0; iTrial < nTrials; iTrial++) {
    // Perform and time the transposition operation
    const double t0 = omp_get_wtime();
    Transpose(A, n);
    const double t1 = omp_get_wtime();
    if (iTrial >= skipTrials)
        t[iTrial] = t1-t0; // Record benchmark result
}
```

图 24-3 忽略前两次运行时间的实验耗时统计

示例算法的性能不但在 Intel Xeon Phi 协处理器上对比 Intel Xeon 处理器相形见绌，而且算法的性能的绝对值同样远远不如 STREAM 基准测试（STREAM 基准测试套件是内存带宽性能的行业参考）。在基准测试系统上，编译和运行 STREAM 的方法及源码见 24.9 节。测试结果中，处理器（CPU）带宽性能为 53.5GB/s，协处理器获得了更高的带宽性能 124.8GB/s。

从图 24-2 可以看出，示例算法性能较差的原因是代码串行部分，这意味着即使硬件支持超线程，算法也只能在一个内核上运行一个线程。其结果是，该算法只能利用一小部分硬件内存带宽。

图 24-2 中代码的串行特性虽然容易观察，但是其串行瓶颈在更加复杂的代码中可能更加难以探测。为此，我们构建了 Intel VTune Amplifier XE（简称“VTune”）系统，目的是在多线程环境中检测算法的串行化特征。

本章中，VTune GUI（图形用户界面）用来收集和分析性能数据。在 Linux 系统中，VTune GUI 通过命令“amplxe-gui”打开。在试验中，根据喜好或者操作环境不同，用户同样可以选择 VTune 命令行界面（见 29.4 节最后一条文献）。

VTune 中在“Projects”标签下存放了应用程序分析设置及结果数据。图 24-4 展示了 VTune GUI 中一个简单项目的项目属性。其中“Application”表示可执行程序的名称。“Application parameters”的值“4000 1000”是命令行参数，指定了矩阵大小 $n = 4000$ 和实验次数为 1000。程序必须运行足够长时间才能有利于收集更高质量的性能数据，因此需要让程序运行大量的次数。使用环境变量将 Intel OpenMP 库引入到系统中，使系统实现 OpenMP 线程在对应内核上的映射并使用同逻辑内核一样多的线程。

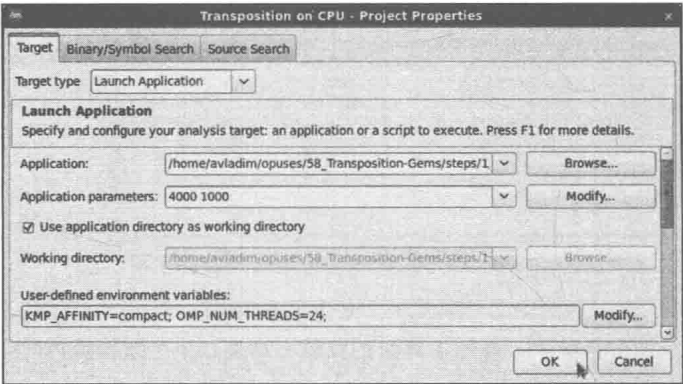


图 24-4 VTune 中配置项目“Transposition on CPU”

项目完成配置后，用户从事先配置的 Analysis Type 中选择一个类型，并运行程序（见图 24-5）。特别有用的分析类型是 General Exploration，它可以启用热点探测及其他一般性能指标。单击 Start 按钮启动该程序，程序执行完成后，实验结果和性能分析在同一个窗口中展示给用户。

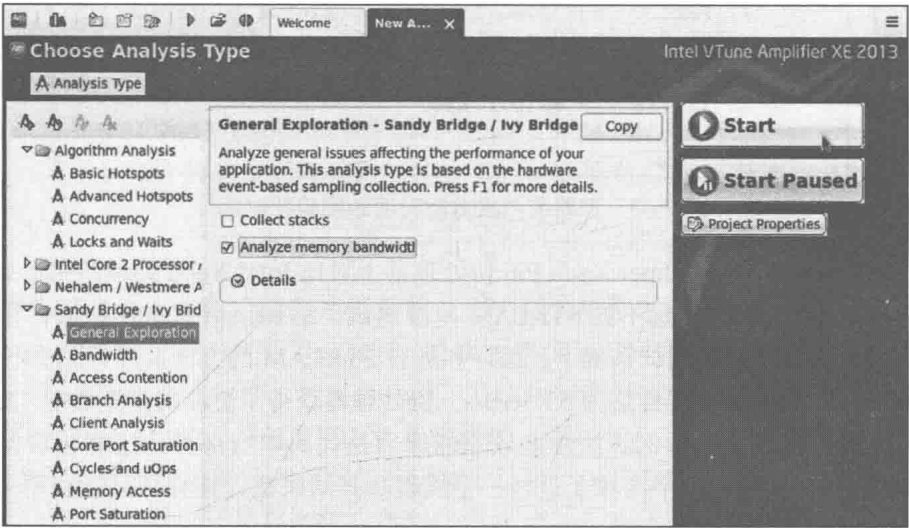


图 24-5 在 Sandy Bridge CPU 架构下启动 General Exploration

图 24-6 展示了结果界面和可选的（如“Summary”“Bottom-up”“Top-down Tree”等）视图。用户通过单击窗口顶部的按钮，选择相应的视图。

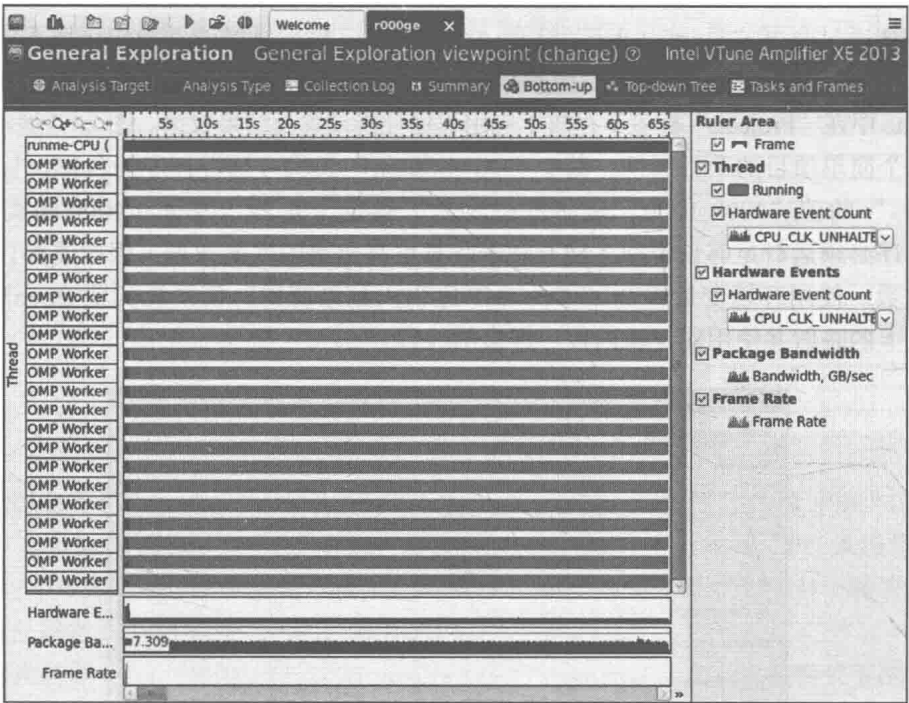


图 24-6 Bottom-up 视图分析结果，显示了算法的负载不均衡。单个线程执行所有工作，其他线程空闲

单击“Bottom-up”视图弹出线程面板，其中显示了处理器 24 个逻辑内核中每个内核

的负载随时间线的变化情况（见 Thread 面板截图）。24 个 Worker 条纹中只有一个存在黑色阴影区域，表明逻辑内核中只有一个有效的工作负载。对于既包含串行又包含并行代码的算法来说，性能图案可能稍许不同（例如黑色条纹带有斑点）。通过使用不同颜色区分串行和并行部分，可以直观地用视觉衡量出串行部分使算法性能降低了多少。

另外，也可以运行特殊的分析类型 Concurrency，以量化并行的使用情况。相应的条目在 VTune 左侧面板的“Algorithm Analysis”文件夹下（见图 24-7）。

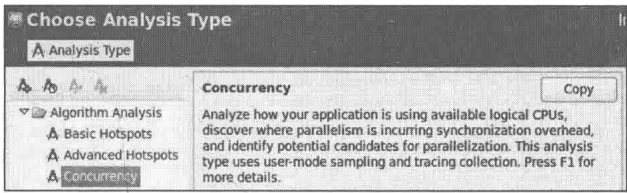


图 24-7 开始并发性分析

图 24-8 通过柱状图显示了在 Summary 视图下的并发性分析结果，显示了一个 x 值为 1 的条形框，即在全部分析时间中只有 1 个线程在运行。对于更加复杂的程序，可能会出现多个条形框，分别显示对应的并行阶段运行时间。

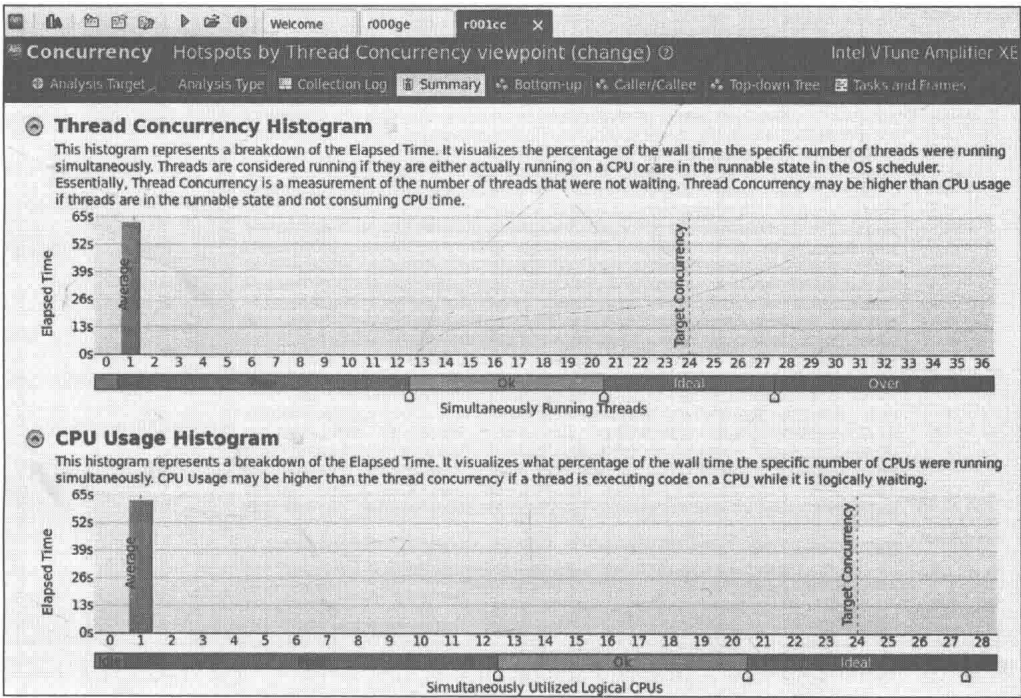


图 24-8 Summary 视图下的并发性分析结果，表明并行利用率差

24.4 并行：使用 OpenMP 增加并行度

既然 VTune 已经广泛用于识别串行性，现在就该用 VTune 来解决矩阵转置性能低的问题了。OpenMP 可以工作在 Intel Xeon 处理器和 Intel Xeon Phi 协处理器中，使用 OpenMP 可以很容易实现程序的并行化。从图 24-2 中可以发现，每次 j 值的迭代都是相互独立的，因此可以在 j 值上使用并行线程分配 for 循环。如图 24-9 所示，可以简单通过在循环语句之前

添加一个 “#pragma OMP” 声明实现循环的并行化。

```
#define FTYPE double

void Transpose(FTYPE* const A, const int n) {
#pragma omp parallel for schedule(static)
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < j; i++) {
            const FTYPE c = A[i*n + j];
            A[i*n + j] = A[j*n + i];
            A[j*n + i] = c;
        }
    }
}
```

图 24-9 方阵就地转置的并行实现

图 24-9 中的代码是实验过程的第 2 步，本书提供了其对应的源码实现。在使用并行代码重新运行基准测试后，发现带宽性能在 CPU 上为 28.6GB/s，在协处理器上为 20.0GB/s(如图 24-27 中的 “Parallel” 标签所示)，可见并行代码带宽性能相对于串行版本有显著提升。

图 24-10 和图 24-11 分别显示了 General Exploration 和 VTune 并发性分析的输出。

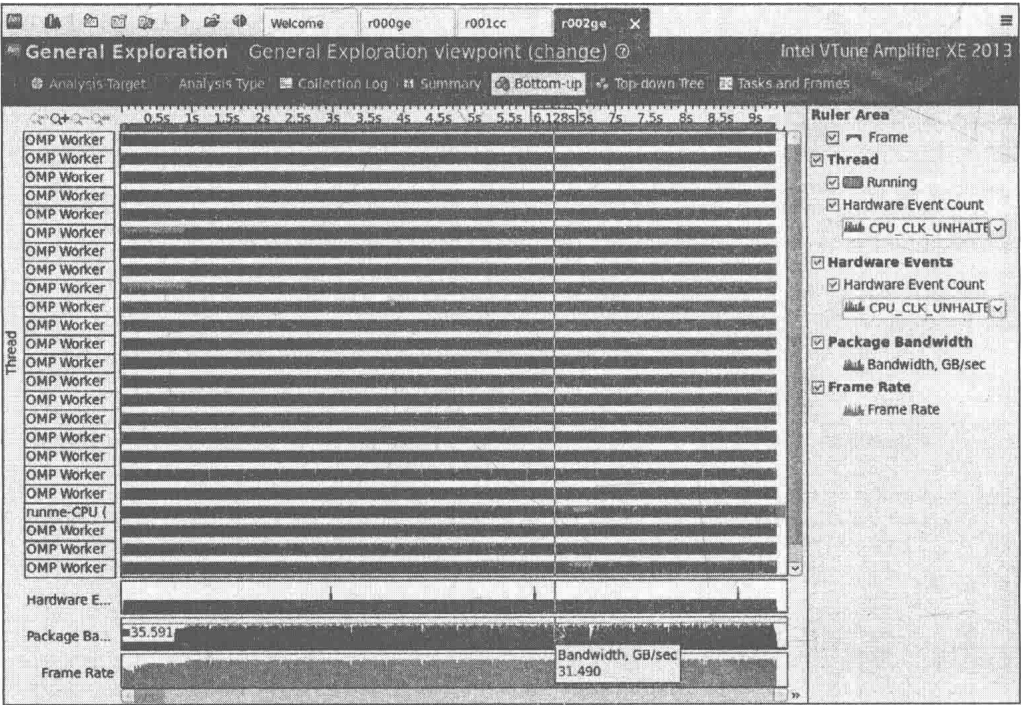


图 24-10 Bottom-up 视图下 General Exploration 分析结果显示多个 CPU 内核得到充分利用

图 24-10 中时间线充满了黑色阴影，这表明所有的内核中都存在有效的工作负载。另外单个内核的时间线比并行代码的时间线更短（见 x 轴时间标签）。

同理，图 24-11 所示线程并发直方图（顶部条形图）中，仅在值为 24 的位置有一个条形框，这证实了所有的 24 个内核都运行了 100% 的有效时间。然而，CPU 使用情况直方图有多个条形框，大部分聚焦在 Ok 区域，这表明内核具有大量的空闲时间。性能分析表明使用优化策略，有可能会进一步提升代码性能。

进一步提升转置代码性能需要更微妙的优化手段，这将在本章的后续部分继续讨论。

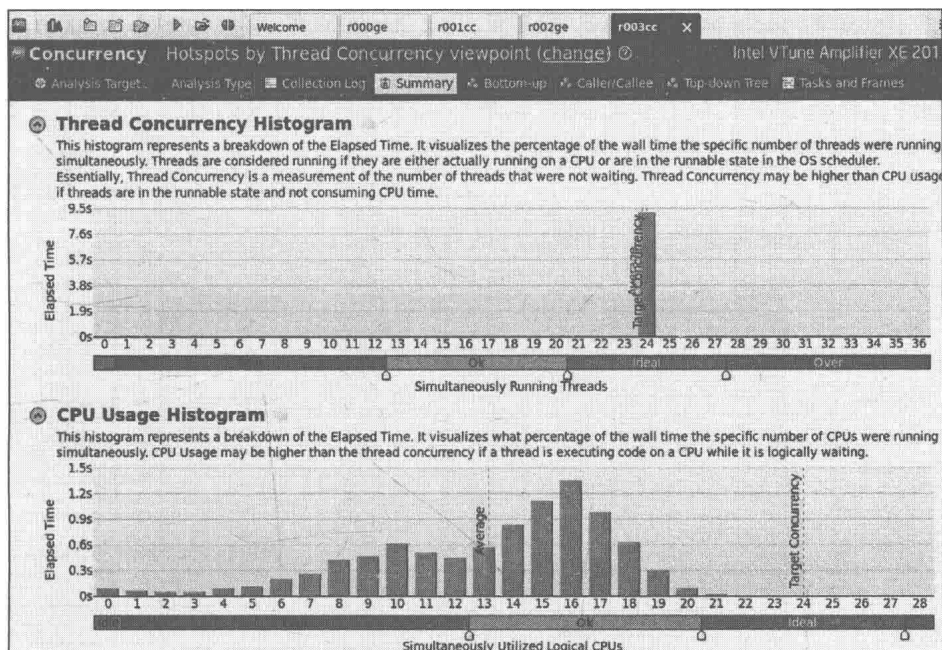


图 24-11 Summary 视图下并发性分析结果，第一幅图展示了算法的高并行性，第二幅图展示了低 CPU 利用率的算法，其大量时间内空转是空闲的

24.5 分块：提高数据局部性

相对于串行代码，通过增加并行性可以显著提高代码的性能，但是并行算法在协处理器上的性能依旧不能让人满意，其性能和处理器上的性能差距达 1.4 倍。

为了提升协处理器上算法的性能，我们需要在 Intel Xeon Phi 协处理器上运行转置代码并使用 VTune 收集算法的性能剖析信息。

在此之前，必须配置一个 MIC 架构的 VTune 项目“Transposition on MIC”。图 24-12 显示了为 Intel Xeon Phi 协处理器的执行配置本机 VTune 项目的一种方法。“Application”中可执行的应用程序是 SSH 的客户端“ssh”。“Application parameters”中 mic0 是 ssh 试图登录的协处理器的主机名。参数“~/runme-MIC”指定进行基准测试的本机协处理器可执行文件所在的路径。命令行参数“4000 1000”分别指定矩阵大小及程序运行次数。

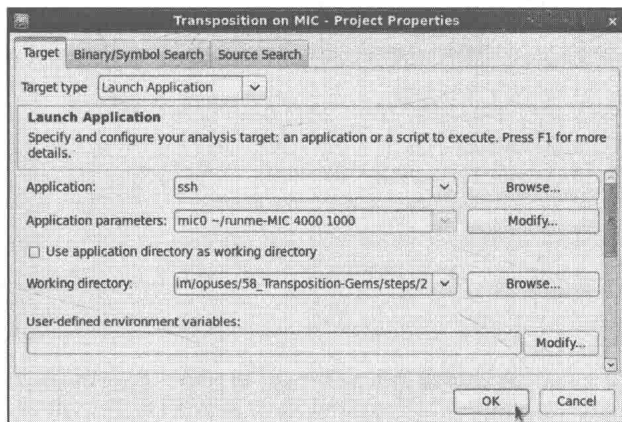


图 24-12 配置 VTune 项目剖析本机协处理器应用。SSH 客户端作为应用程序，可执行文件作为其命令行参数

注意 Intel VTune Amplifier XE 可以对 Intel Xeon 处理器和 Intel Xeon Phi 协处理器上运行的程序进行性能分析。对于第一代 Intel Xeon Phi 协处理器，选择“Knights Corner Architecture”分析类型。对于本机 MIC 程序，同样可以使用应用程序“ssh”并作为命令行参数传递需要测试的可执行文件的名称。

此外，如图 24-13 所示，为了在 VTune 中能够看到函数名称和源代码，对于 MIC 架构，我们必须为 VTune 指定包含二进制文件的目录名称。

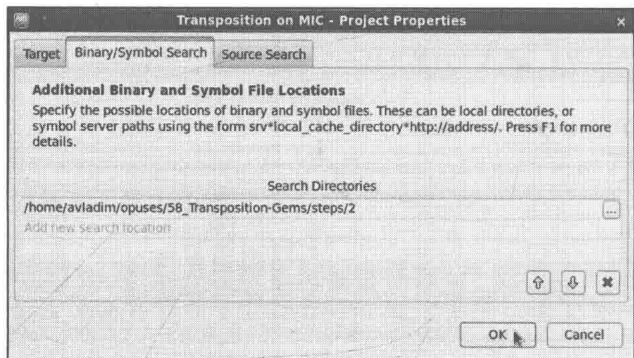


图 24-13 在 VTune 项目中，为了能够观察正在运行的函数名称及源代码，使用参数“-g”编译源代码，并为 VTune 指定主机中包含可执行文件的目录

本机模式下，运行 VTune 分析器之前，用户必须复制可执行文件和所有依赖库到协处理器中。本示例中，唯一需要复制到协处理器中的依赖库是 Intel OpenMP 库。例如，我们将依赖库复制到协处理器的 mic0:/lib64/ 目录下。相应地，也可以使用 NFS 共享使处理器和协处理器共享相应的库、可执行文件及数据。更多信息请参考 24.9 节第 9 条文献。

配置项目且本机应用就绪后，分析程序即可启动。和主机一样，General Exploration 分析是一个很好的起始点，虽然这一次是在 KnightsCorner 平台上（即在 Intel Xeon Phi 协处理器上分析），如图 24-14 所示。

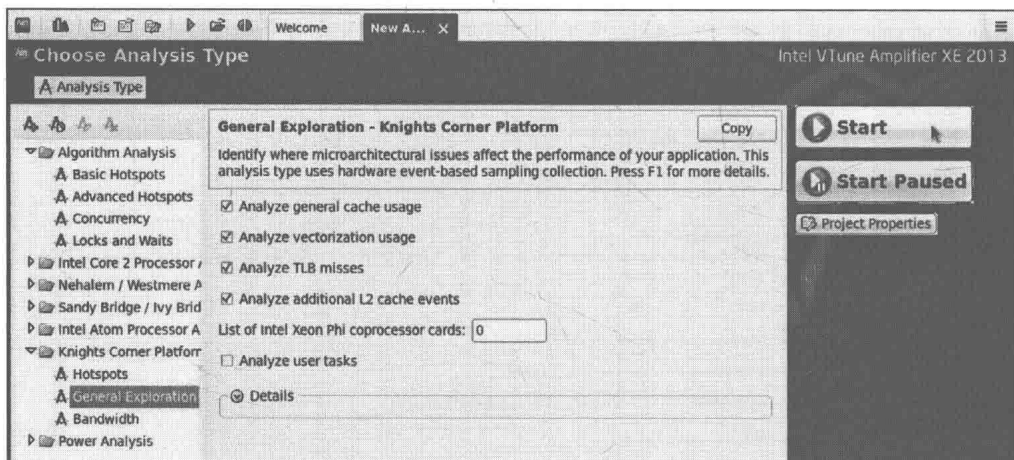


图 24-14 在 Intel Xeon Phi 协处理器下的 VTune 项目中，启动 General Exploration 分析

图 24-15 所示是 Summary 视图下的性能剖析结果。通过突出显示的性能指标，我们可以看出 VTune 在该视图下的强大功能。这里突出显示了 CPI Rate 和 Estimated Latency Impact。

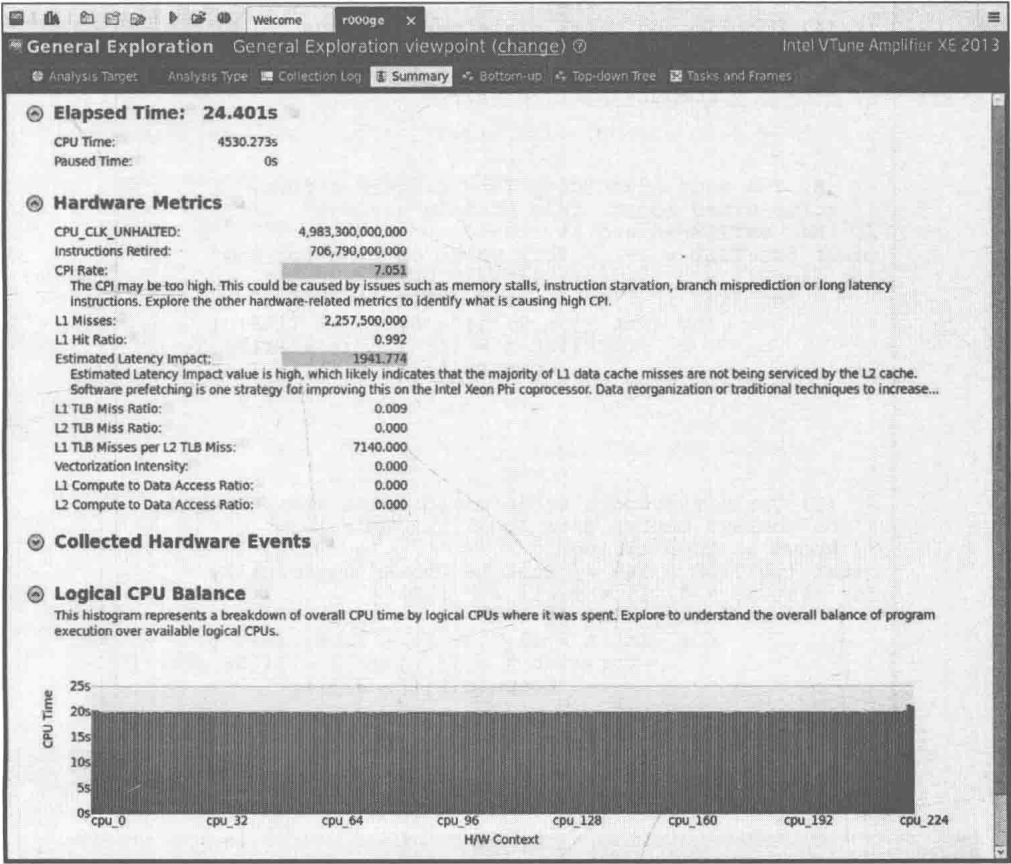


图 24-15 在 Intel Xeon Phi 协处理器下的 VTune 项目中，General Exploration 分析总结

CPI 指标显示执行每条指令平均需要多少个周期，它衡量延迟对性能的影响。在 Summary 视图下，CPI 针对的是单个线程上的算法执行。在 Intel Xeon Phi Knights Corner 处理器中，每个内核具有 4 个线程，且每个内核都是顺序双发射的，因此其 CPI 理论最小值是 2.0。高 CPI 值表示应用中的时间延迟较大，但对于以内存访问带宽为瓶颈的应用（如矩阵转置），这 CPI 的影响较小。事实上，VTune 中 MIC 架构下内存带宽高度优化的 STREAM 基准测试显示的 CPI 率大于 8.2。

如图 24-15 所示，Estimated Latency Impact 是另一个 VTune 突出显示的指标。该指标在本示例中非常重要。Estimated Latency Impact 可以通过统计特定的数据得到。特定的数据是指，一级缓存未命中，二级缓存中未缓存，只能从主存中读取的数据。VTune GUI 中提示了有可能提升算法性能的解决方法。本示例中，为了提升软件整体性能，VTune 建议程序员提升算法预取并增加数据局部性访问。

本示例适合采用提高数据局部性的方法。传统的、简单而有效的局部性优化方法即是分块。分块法需要对内存访问进行重新排序，一旦数据被取入到缓存中，需要更加及时地在一般循环算法中进行数据重用。

从实际的编程角度来看，具有两个嵌套循环的算法分块涉及取数据循环和置换循环问题，这里通过将分块循环外置的方法予以避免。图 24-16 对该过程进行了描述，三个代码段分别是（A）普通嵌套循环，（B）分块循环准备，（C）分块循环（通过置换两个循环得到）。


```

// (A) Unoptimized case: plain nested loops
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        Compute(A[i], B[j]);
    }
}

// (B) The same exact code in a different form:
// strip-mined loops. This example assumes
// that m%TILE==0 and n%TILE==0.
const int TILE = 1; // This value is fine for now
for (int ii = 0; ii < m; ii += TILE) {
    for (int i = ii; i < ii + TILE; i++) {
        for (int jj = 0; jj < n; jj += TILE) {
            for (int j = jj; j < jj + TILE; j++) {
                Compute(A[i], B[j]);
            }
        }
    }
}

// (C) Optimized code: strip-mined loops are permuted
// to achieve better data locality. This is
// known as loop tiling.
const int TILE = 16; // Must be chosen empirically
for (int ii = 0; ii < m; ii += TILE) {
    for (int jj = 0; jj < n; jj += TILE) {
        for (int i = ii; i < ii + TILE; i++) {
            for (int j = jj; j < jj + TILE; j++) {
                Compute(A[i], B[j]);
            }
        }
    }
}

```

图 24-16 循环分块

在普通的循环算法 (A) 中, 在变量 j 中进行内层循环迭代, 对应的元素由 $B[j]$ 取得。假设缓存的大小为 n , 则 A 和 B 中元素的大小使得数组 A 与 B 不适合缓存。因此当 j 从 0 到 $(n-1)$ 迭代 j 的时候, 元素 $B[0]$ 则不存在于缓存中。当算法中增加 i 值并在 j 为 0 时重新开始, 则处理器需要重新从主存中读取 $B[0]$, 导致了性能损失。与此相反, 在分块循环 (C) 中, $B[0]$ 的值在 $TILE$ 迭代后可重用, 假定 $TILE$ 足够小, 则 $B[0]$ 依旧会存储在缓存中, 并被命中。

在算法优化过程中, 针对不同计算机架构和算法, $TILE$ 的值是变化的, 且 $TILE$ 值应该足够小, 使数据访问能够持续命中。但是 $TILE$ 值也有可能有限制 (若在 j 上的内层循环向量化, 则最适用于 $TILE$ 的值应该是 SIMD 向量长度的整数倍)。例如双精度计算过程中, Knights Corner 架构的向量长度是 8, 则 $TILE$ 的值必须是 8 的倍数。

图 24-9 展示了矩阵转置代码, 算法分块时必须考虑其他的一些情况。例如图 24-9 中内层循环的上限取决于外层循环中的变量值, 因此分块算法必须确保条件 ($j < i$)。其次, 如果无法保证矩阵大小 n 为 $TILE$ 的整数倍, 分块算法必须要保证矩阵数据访问不超界。

综上所述, 图 24-17 展示了矩阵转置算法中的分块版本。图 24-27 中标记为 “Tiled” 的条形框显示了该算法的性能。虽然在 CPU 平台上性能提升不是很明显, 但在协处理器平台性能提升达 30%。

在 VTune 系统中, General Exploration 分析确定了使用分块可以提升算法在协处理上的性能 (见图 24-18)。在 Summary 视图下, Estimated Latency Impact 显示的是 1082, 而不是优化前的 1942 (见图 24-15)。由于在协处理器上获得的性能仍然低于主机处理器, 而且算

法只达到 STREAM 基准测试带宽的 20%，进一步优化仍然是必要的。下一节将继续讨论其他的 VTune 分析和代码优化方法。

```
const int TILE = 16; // Empirically chosen tile size

#pragma omp parallel for schedule(static)
for (int ii = 0; ii < n; ii += TILE) { // Tiling
    for (int jj = 0; jj <= ii; jj += TILE) { // Tiling

        // Do not go beyond matrix border
        const int jMax = (jj+TILE < n ? jj+TILE : n);

        for (int j = jj; j < jMax; j++) {

            // Do not go beyond main diagonal
            const int iMin = (ii > j ? ii : j+1);

            // Do not go beyond matrix border
            const int iMax =
                (ii+TILE < n ? ii+TILE : n);

            // Transpose the tile
            for (int i = iMin; i < iMax; i++) {
                const FTYPE c = A[i*n + j];
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
        }
    }
}
```

图 24-17 就地方阵转置算法的分块实现

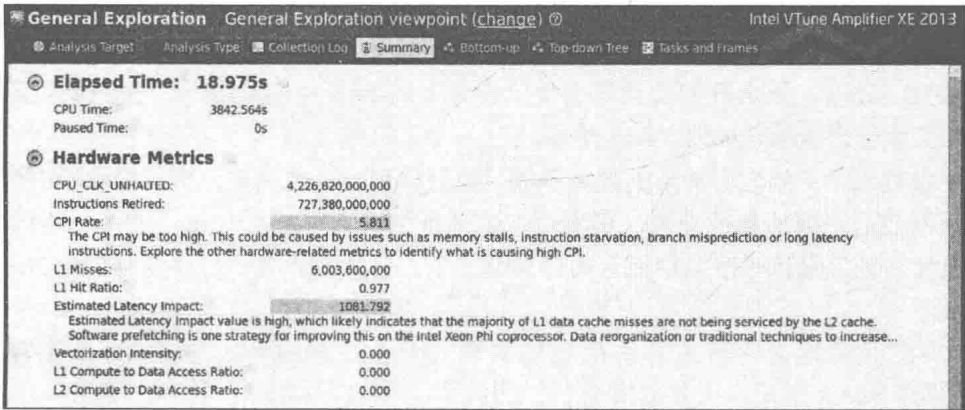


图 24-18 分块代码在协处理器上的 General Exploration 分析结果显示，延时虽然有所减少，但 Estimated Latency Impact 仍然很高

24.6 规范化：多版本微内核

VTune 有一个非常实用的功能：能够将针对整个代码程序的分析，细化到每一行代码，甚至是每一条汇编指令。在下面两种情况下，这个功能将会非常有用：

- 1. 在复杂的程序中，它可以让程序员找到最耗时的函数和代码行（热点）。
- 2. 已经知道“热点”在哪里后，用户可以查看对应的汇编代码（在 Intel C++ 编译器的编译指令中增加“-S”选项，就可以生成汇编代码清单）。然而，查看汇编代码的过程是非常困难的，因为对于循环等语句，编译器会生成多个不同的分支，我们很难分辨哪个分支是在运行时执行的。VTune 中带注释的汇编代码清单可以很清楚地告诉我们哪一段代码分支执

行了，因为在每一行汇编代码的旁边都会有其统计到的执行时间（如图 24-19 所示）。

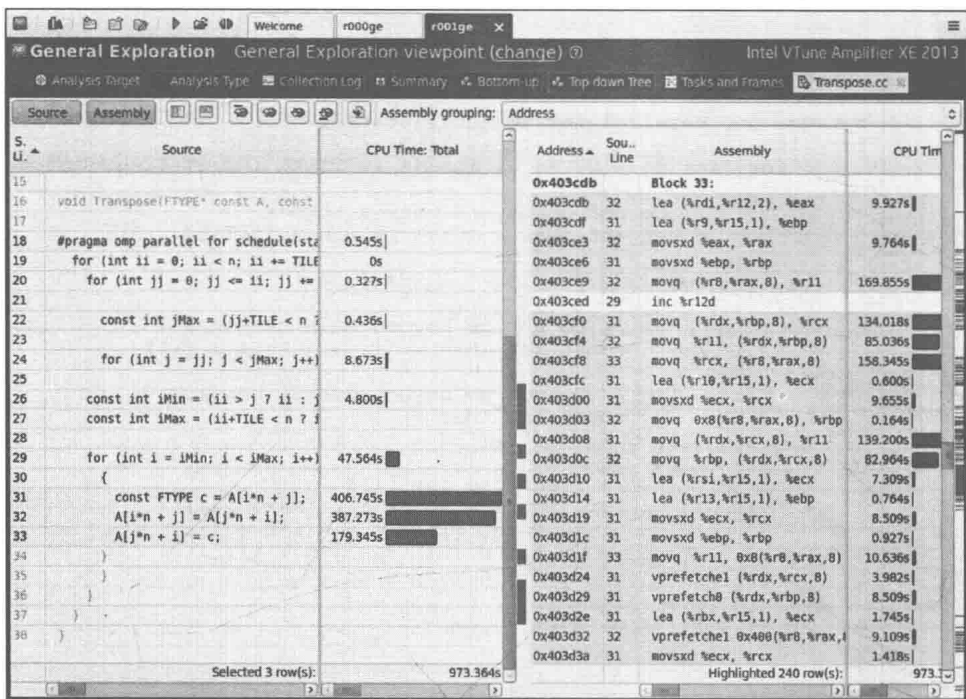


图 24-19 VTune 允许用户查看每一行源代码或汇编代码的事件计数。这样就可以识别出哪些代码分支执行得最频繁

为了查看带注释的汇编代码清单，程序员需要打开“Bottom-up”视图，并且双击所感兴趣的函数。此时，该函数的源代码会展示出来，而单击“Assembly”按钮就可以查看这个函数的汇编代码清单，如图 24-19 所示。

即使没有关于汇编代码清单的详细分析，也可以很容易地发现，执行的代码中许多指令都有涉及行与列之间的数据交换。事实上，编译器引入了一系列“lea”指令，用于获取操作数的地址。该汇编代码清单中还有内存复制指令，但并非所有的内存复制指令都用于矩阵数据交换。

图 24-17 是就地方阵转置算法的分块实现。需要注意的是，内层循环的循环计数从 iMIN 开始到 iMAX 结束。大多数情况下，(iMAX-iMIN) == TILE，其中，TILE == 16。然而，有时（靠近矩阵主对角线或边缘时），循环计数将有可能小于 16。图 24-17 中的代码通过增加边界判断，实现了适用于多版本的内层循环，这样它就可以处理任意大小的矩阵。然而，在这种情况下，可执行代码的通用性往往会导致应用程序性能降低。

但是，程序员可以在刚开始编写代码时就考虑到多版本情况，这样可以帮助编译器生成高效的可执行代码，其中包含主要的数据转移指令。编码时考虑到多版本情况后，我们可以将矩阵分为三个区域（如图 24-20 所示）。

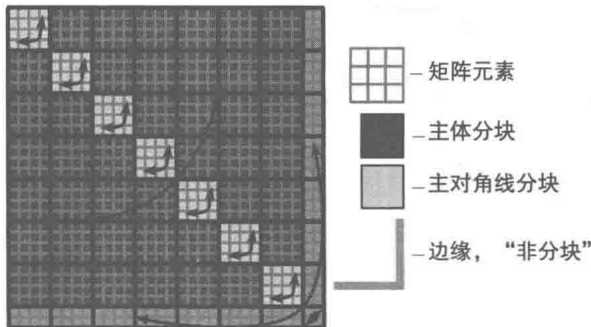


图 24-20 将矩阵分为三个区域，以消除边界判断

(a)“主体”，其中的 16×16 分块可以在没有任何范围检查的情况下进行数据转置。

(b)“主对角线”，处理其中的 16×16 分块时需要将微内核稍加修改。

(c)“边缘”，这是围绕矩阵边界的部分，对于这部分，需要用各自的“非分块”微内核来处理。然而，仅当矩阵大小不是 $TILE == 16$ 的整数倍时，这个区域才会存在。

矩阵分割后，处理不同区域的数据转置的可执行代码都简化了。这种方法的 C 语言代码实现如图 24-21 所示。值得注意的是，第一个 OpenMP 并行循环中，内层 j 循环的循环次数是可以在编译时确定的，即与分块大小相等。这使得编译器可以实现更高效的数据转置代码。

```
const int TILE = 16; // Empirically chosen tile size
// nEven is a multiple of TILE
const int nEven = n - n%TILE;
// Complete tiles in each dimension
const int wTiles = nEven / TILE;

#pragma omp parallel
{
    // The main body tile transposition microkernel
    #pragma omp for schedule(static)
    for (int ii = TILE; ii < nEven; ii += TILE)
        for (int jj = 0; jj < ii; jj += TILE)
            for (int j = jj; j < jj+TILE; j++)
                for (int i = ii; i < ii+TILE; i++) {
                    // Key to performance: the bounds of
                    // this loop are known at compile time
                    const FTYPE c = A[i*n + j];
                    A[i*n + j] = A[j*n + i];
                    A[j*n + i] = c;
                }

    // Transposing the tiles along the main diagonal
    #pragma omp for schedule(static)
    for (int ii = 0; ii < nEven; ii += TILE) {
        const int jj = ii;
        for (int j = jj; j < jj+TILE; j++)
            for (int i = ii; i < j; i++) {
                const FTYPE c = A[i*n + j];
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
    }

    // Transposing ``peel'' around the perimeter
    #pragma omp for schedule(static)
    for (int j = 0; j < nEven; j++)
        for (int i = nEven; i < n; i++) {
            const FTYPE c = A[i*n + j];
            A[i*n + j] = A[j*n + i];
            A[j*n + i] = c;
        }
    } // End of OpenMP parallel region

    // Transposing the bottom-right corner
    for (int j = nEven; j < n; j++)
        for (int i = nEven; i < j; i++) {
            const FTYPE c = A[i*n + j];
            A[i*n + j] = A[j*n + i];
            A[j*n + i] = c;
        }
}
```

图 24-21 多版本、分块的就地方阵转置算法实现

实际上，多版本代码的基准测试表明：在主机 CPU 上是 29.4GB/s，在协处理器上是 81.2GB/s。在主机上，性能只比之前的版本稍有提升，但是，考虑到还存在 1GB/s 左右的标准差（这里未显示），所以可以说多版本代码的 GPU 性能相对优化前没有显著提升。但是

对于协处理器，情况就大不一样。因为在协处理器上，多版本的代码相对之前版本获得了 3 倍的性能提升。

从 VTune 中的编译代码（见图 24-22）可以看出，主体块的指令流显得更有规律。大部分时间被取用指令“movq”所占用，指令“movq”可用于复制四字数据。

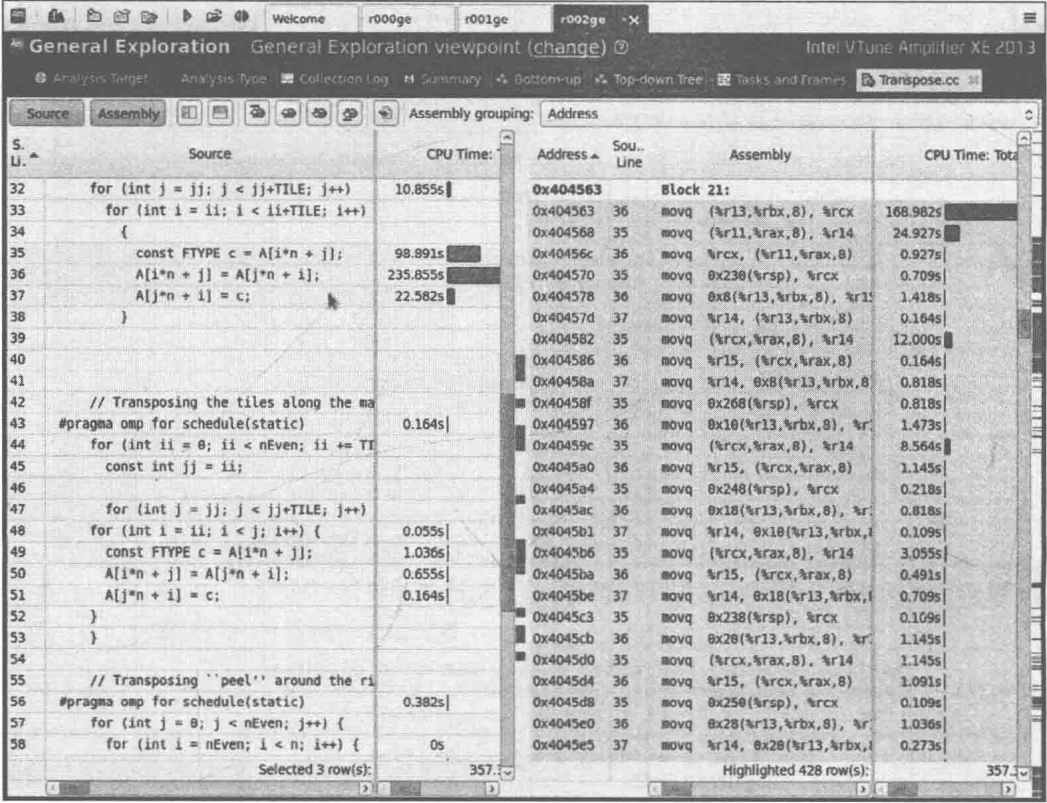


图 24-22 矩阵划分为三个区域后的矩阵转置编译代码清单。程序员完成转置微内核的多版本实现后，编译器产生的指令流变得更有规律了

多版本优化的代码可能与鼓励抽象的代码风格相违背。然而，这种做法在高性能计算应用中非常普遍。实际上，图 24-17 中的代码相比图 24-21 中的代码紧凑得多。此外，图 24-21 中的代码可能也是不符合“教科书”中的做法的。因为它存在大量冗余代码。然而，3 倍的显著性能提升，为冗余代码的引入提供动力和理由，因为这些冗余代码可以协助编译器产生更高效的可执行代码。

指导这种情况的优化“经验法则”是使影响性能的关键部分（内层循环）尽可能越简单。举例来说，数组边界最好在编译时就加以确定。此外，如果采用矢量方法，则循环计数应为 16 的倍数。

在最后一个优化步骤中，我们将再次使用 VTune 来诊断并解决关于并行性不足的性能瓶颈。

24.7 预组织：释放更多的并行性

在 VTune 性能分析的帮助下，一个很容易想到的矩阵转置代码优化方法就是“预组织”程序的并行粒度和分配。

如图 24-23 所示，多版本代码的大多数 CPU 时钟周期都消耗在 libiomp5.so 上，libiomp5.so 是 Intel OpenMP 库的一部分。

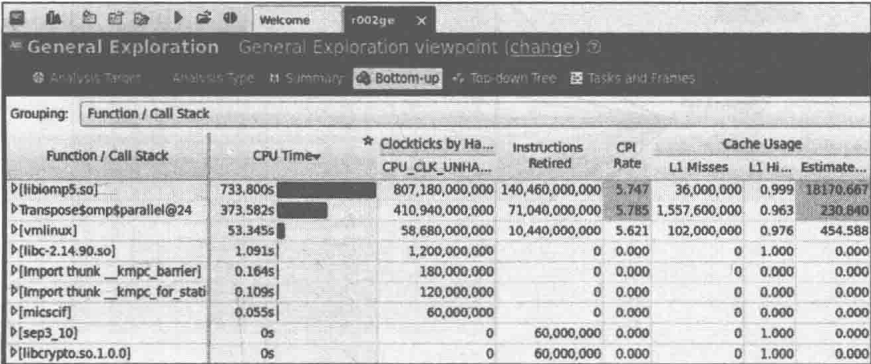


图 24-23 图 24-21 中代码 CPU 占用的“Bottom-up”视图。Intel OpenMP 库函数占用大量运行时间

在 OpenMP 库上花费过多的时钟周期预示着存在大量的闲置线程，而线程闲置的原因可能是负载不均衡或者同步所造成的串行化。纵观图 24-21 中的代码，可以发现两个问题：

- 1. 物理线程间负载不均衡。转置函数中的第一个并行 for 循环的迭代次数太少了。对于 $n = 4000$ 和 $TILE = 16$ ，该循环也只有 $4000/16 = 250$ 次迭代。Intel Xeon Phi 上有 240 个硬件线程，仅仅 250 个并行线程很难实现物理线程间的负载均衡。
- 2. 线程本身的负载也是不均衡的。循环索引变量 ii 越小，对应的迭代（即所对应的物理线程）的工作量也越小。这是由于第二层循环的终止条件 ($jj < ii$) 和 OpenMP 语句“schedule (static)”。即使将循环调度从“static”（所有线程分配相同的迭代次数）换为“dynamic”或“guided”（迭代根据线程运行状态来进行分配），也会因为问题 1 而难以实现负载平衡。

因为 Intel Xeon Phi 协处理器上有大量的可用硬件线程，所以使得并行性不足成为使用 MIC 架构时所面临的典型挑战。解决方案是减小并行粒度，为编译器和 OpenMP 运行时库释放更多的并行性。

对于矩阵转置，工作分配的粒度是一行块，块的数量取决于 ii 的值（如图 24-24 左图所示），对应的工作项数量为 $O(n/TILE)$ 。解决办法是缩小工作分配的粒度到单个块（如图 24-24 右图所示），使工作项的数量提升为 $O((n/TILE)^2)$ 个。对于 $n = 4000$ 和 $TILE = 16$ ，总共有 62500 个工作项，这足够让 240 个物理线程都一直处于忙碌状态。

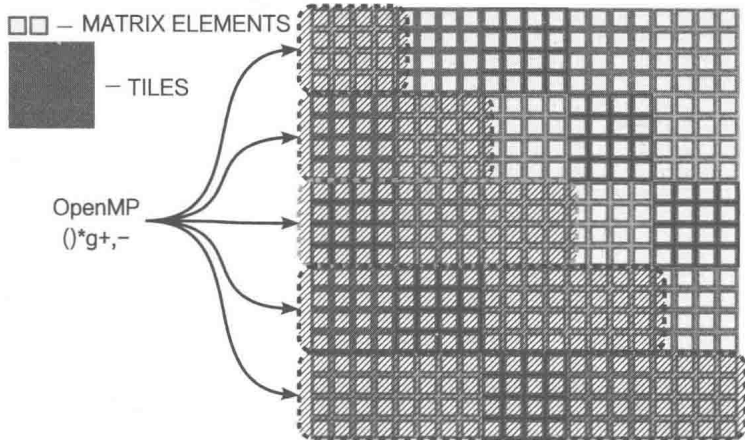


图 24-24 释放转置代码的并行性。用单个分块的工作分配粒度，代替整行的工作分配粒度

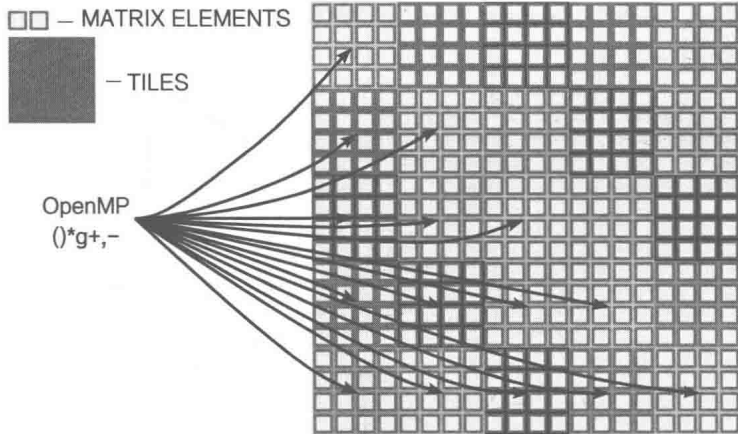


图 24-24 (续)

为了改变工作分配的粒度，需要创建一个贯穿整个并行空间的单一索引到块位置的映射。根据图 24-21 所示的分块枚举过程可知，映射过程可能需要涉及整数除法和取模运算。一个比较普遍的解决办法是预先计算此映射（预组织）。提前计算有两个好处，首先，它可以消除整除和取模运算带来的延迟。其次，它具有可扩展性，即可以简单地扩展到其他更复杂的矩阵遍历模式（例如，24.9 节第 1 条文献）。

“预组织”的转置代码如图 24-25 所示。需要注意的是，在调用主函数 Transpose 之前，需要调用函数 CreateTranspositionPlan 创建“组织”数组。在最后一次调用 Transpose 函数结束后，调用函数 DestroyTranspositionPlan 来释放“组织”数组空间。

```
const int TILE = 16; // Empirically chosen tile size
void CreateTranspositionPlan(int* & plan, const int n) {
    // This function must be called prior to
    // calling the function Transpose()
    // to allocate and fill the array plan[],
    // which contains the tile traversal order.

    // Number of complete tiles in each dimension
    const int wTiles = n / TILE;

    // Number of complete tiles below the main diagonal
    // in the whole matrix
    const int nTilesParallel = (wTiles-1)*wTiles/2;

    // Order of tile traversal
    plan = (int*)
        malloc(sizeof(int)*(2*nTilesParallel));

    // Storing tile locations in the plan
    int c = 0;
    for (int ii = 1; ii < wTiles; ii++)
        for (int jj = 0; jj < ii; jj++) {
            plan[2*c + 0] = ii*TILE;
            plan[2*c + 1] = jj*TILE;
            c++;
        }
}
```

图 24-25 “预组织”的转置代码：扩展并行迭代空间

性能测试证实，扩大迭代空间和提前计算“组织”能够显著提升转置代码的性能（见图 24-26）。在主机 CPU 上，获得了 40% 的性能提升，达到了 41.4GB/s，而在协处理器上，获得了 7% 的性能提升，达到 87.0GB/s（见图 24-27）。


```
void DestroyTranspositionPlan(int* plan) {
    // Free the memory allocated by
    // the function CreateTranspositionPlan()
    free(plan);
}

void Transpose(FTYPE* const A, const int n, const int*
const plan) {

    // nEven is a multiple of TILE
    const int nEven = n - n%TILE;

    // Complete tiles in each dimension
    const int wTiles = nEven / TILE;

    // Numer of tiles in the matrix body
    const int nTilesParallel = wTiles*(wTiles - 1)/2;

#pragma omp parallel
{
    // Distribute work with single tile granularity
#pragma omp for schedule(static)
    for (int k = 0; k < nTilesParallel; k++) {
        // For large matrices, most of the work
        // takes place in this loop.

        // Pull location of the tile from the plan
        const int ii = plan[2*k + 0];
        const int jj = plan[2*k + 1];

        // The main tile transposition microkernel
        for (int j = jj; j < jj+TILE; j++)
            for (int i = ii; i < ii+TILE; i++) {
                const FTYPE c = A[i*n + j];
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
    }

    // The rest of the code in function Transpose(...)
    // is the same as in Figure 21. That code
    // is required to transpose the main diagonal tiles
    // and the "peel" around the matrix perimeter
    ...
}
```

图 24-25 （续）

General Exploration General Exploration viewpoint (change) ⓘ

Analysis Target Analysis Type Summary Bottom-up Top-down Dec Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	★ Clockticks by Ha...		Instructions Retired	CPI Rate	Cache Usage		
		CPU	CLK UNHA...			L1 Misses	L1 HI...	Estimate...
↳ Transpose\$omp\$parallel@24	605.945s		666,540,000,000	66,960,000,000	9.954	1,422,900,000	0.970	425.801
↳ [libiomp5.so]	447.764s		492,540,000,000	89,580,000,000	5.498	24,000,000	0.999	17235.500
↳ [vmlinux]	42.927s		47,220,000,000	4,440,000,000	10.635	0	1.000	0.000
↳ main\$omp\$parallel_for@19	3.655s		4,020,000,000	0	0.000	0	0.000	0.000
↳ [libc-2.14.90.so]	0.818s		900,000,000	180,000,000	5.000	0	1.000	0.000
↳ [import thunk __kmpc_barrier]	0.109s		120,000,000	0	0.000	0	1.000	0.000
↳ [ld-2.14.90.so]	0.055s		60,000,000	0	0.000	0	0.000	0.000
↳ [import thunk __kmpc_for_stati	0.055s		60,000,000	0	0.000	0	0.000	0.000
↳ [import thunk __kmpc_for_stati	0.055s		60,000,000	0	0.000	0	1.000	0.000
↳ main\$omp\$parallel_for@30	0s		0	120,000,000	0.000	0	0.000	0.000

图 24-26 Intel Xeon Phi 协处理器上图 24-25 所示代码的 General Exploration 分析的 Bottom-up 视图。释放出更多的并行性，使得线程闲置减少，即减少了 libiomp5.so 的耗时，从而获得了性能上的显著提升

分析 VTune 的性能数据，我们可以发现，libiomp5.so 所占时间大大减少，且现在的“热点”是 Transpose 函数中的并行循环，这证明该优化步骤达到了预期效果。

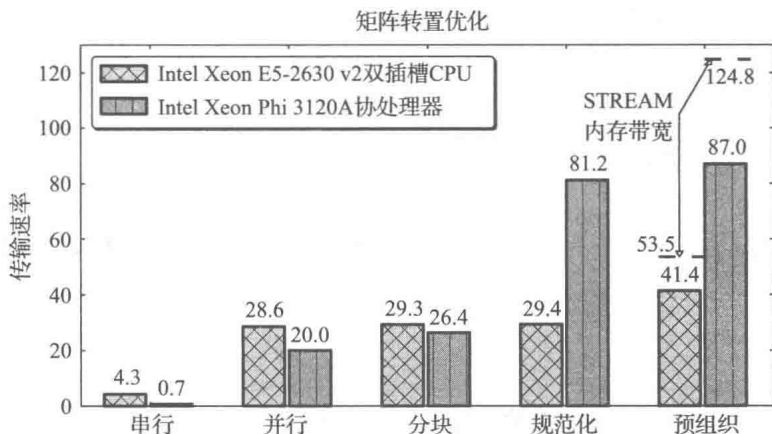


图 24-27 在 Intel Xeon CPU 和 Intel Xeon Phi 上，矩阵转置函数在不同优化阶段的性能图。性能的理论峰值是 STREAM 内存带宽，如虚线所示

24.8 总结

剖析指导的就地矩阵转置函数的优化过程已经完成，优化结果在图 24-27 中汇总。现在回顾本章中执行的优化步骤。

1. 矩阵转置函数最开始是用简单的串行代码实现的（见图 24-2），但是这个版本的代码在 Intel Xeon 处理器上以及在 Intel Xeon Phi 处理器上的性能都很差（见图 24-27 中的“串行”条形框）。用 VTune 分析并发性，我们可以发现，缺乏并行性是性能糟糕的原因。

2. 通过 OpenMP 的编译制导语句实现并行化（如图 24-9 所示），性能显著提升，但还远未发挥出 MIC 架构（见图 24-27 中的“并行”条形框）的能力。协处理器上 VTune 的 General Exploration 分析结果显示是访存延迟导致的这一情况。

3. 通过循环分块（见图 24-17），提高数据局部性，从而降低访存延迟的影响。协处理器上的代码性能得到了提升，但仍落后于 CPU（见图 24-27 中的“分块”条形框）。通过 VTune 观察函数的汇编代码发现，函数的内层循环是在运行时确定矩阵边界的，为了保持通用性，编译器生成的汇编代码不得不很复杂。

4. 多版本 C 语言代码的实现（见图 24-21）解决了内层循环中的这个问题。这使得编译器可以将内层循环的数据转置规范化，从而使 Intel Xeon Phi 协处理器的性能超过了主机（见图 24-27 中的“规范化”条形框）。

5. 多版本代码的“Bottom-up”视图显示，函数的瓶颈在于 OpenMP 库。在 OpenMP 库中花费大量时间的原因是有限的并行迭代空间。因为有限的并行迭代空间会导致负载不平衡，造成线程之间相互等待。通过“预组织”，将并行的粒度从一排块降到单个块（见图 24-25）后，处理器和协处理器（见图 24-27 中的“预组织”条形框）上的这个问题都得到了解决。此时，VTune 的报告显示目前的“热点”是数据转置循环，这证明该优化步骤达到了预期效果。

与 STREAM 基准测试的内存带宽相比，在两个平台（处理器和协处理器）上，最终取得的性能转置代码分别达到了 STREAM 带宽的 77% 和 70%。STREAM 是简单的复合基准测试，用于衡量 4 个简单的向量内核的可持续性内存带宽以及相应的计算速率。精心优化的 STREAMS 带宽通常精确代表平台最大的可持续性内存带宽。所以，能够达到 70% 已经是非常显著的成果。

本应用还可以进一步优化。“tuning knob”可以通过软件预取进一步提高性能。在 Intel Xeon Phi 协处理器上,对于从内存到 2 级缓存的数据转移,软件预取可以作为对硬件预取的补充。但是对于从 2 级缓存到 1 级缓存的数据转移,只有软件预取一种方式。通常,编译器会评估并在可执行代码中实现预取。然而,程序员也可以通过使用 `#pragma (no) prefetch` 和编译器参数 `-opt-prefetch-distance` 修改预取行为。另外,还可以在代码中显式调用预取内置函数来修改预取行为。这种优化是与环境实现相关的,不能直接扩展到 Intel Xeon Phi 系列之后的产品,如下一代代号为 Knights Landing 的产品。

与此相反,基于高级编程语言的可移植的、面向未来的优化方法,使程序在协处理器上相比处理器获得了 2.2 倍的加速比,并且达到了理论最佳性能的 70% 以上。

本章还证明,在优化的 MIC 架构上使用性能剖析工具(如 Intel VTune Amplifier XE)能够一举两得。首先,它可以用于检测应用程序的“热点”(甚至在汇编指令层,识别哪些分支在运行时执行)。其次,VTune 还计算整体性能指标,如时延的影响、缓存命中率、向量指令利用情况等,并提出代码改进方向。

24.9 更多信息

下面是本章中提及的一些扩展阅读资料:

- Vladimirov, A., 2013. Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Colfax Research. <http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx>.
- Frigo, M., Johnson, S., 2005. The design and implementation of FFTW3. *Proc. IEEE* 93(2), 216-231. <http://www.fftw.org/fftw-paper-ieee.pdf>.
- Sung, I.-J., et al., 2014. In-place transposition of rectangular matrices on accelerators. In: *PPoPP*, pp. 207-218. <http://dx.doi.org/10.1145/2555243.2555266>.
- Pen, U.-L., et al., A free, fast, simple and efficient total variation diminishing magnetohydrodynamic code. *Astrophys. J. Suppl.* 149, 447. <http://dx.doi.org/10.1086/378771>.
- STREAM 基准测试集源码: <http://www.cs.virginia.edu/stream/>。
- 在 Intel Xeon Phi 协处理器上编译、执行 STREAM 的指令 <https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad>.
- Colfax International, 2013. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. ISBN-10 0-9885234-1-8|ISBN-13 978-0-9885234-1-8. <http://www.colfax-intl.com/nd/xeonphi/book.aspx>.
- Colfax SXP7450 workstation specifications. <http://www.colfax-intl.com/nd/workstations/sxp7450.aspx>.
- Intel VTune Parallel Amplifier online documentation. https://software.intel.com/en-us/vtuneampxe_2013_ug_lin.

基于 ITAC 的异构 MPI 应用优化

Vadim Karpusenko

美国, Colfax International

本章的重点是在由 Intel Xeon 处理器以及 Intel Xeon Phi 协处理器组成的异构集群环境下讨论消息传递接口 (MPI) 应用程序的负载均衡问题, 并用金融业中亚式期权回报计算作为应用实例。我们将会考虑三种情况: 非均衡的对称 MPI 代码, 基于预先计算出的集群组件性能而实现的手动均衡, 以及“老板-员工”模式 (也称为主/从通信模式) 下的动态负载均衡。

25.1 亚式期权定价

期权是一种允许一方 (称为受益人) 买入 (看涨期权) 或卖出 (看跌期权) 的合同。在将来的某一天 (期权到期), 股票市场的资产会以在合同签订时一致同意的执行价格从另一方流出或流入另一方。购买合同称为看涨期权, 而卖出合同称为看跌期权。不同于期货合同, 期权赋予受益人选择是否行使该交易的权利。这种选择一般由期权到期时资产的市场价格而定。例如, 如果在期权到期日, 该资产的股票市场价格比期权合同中的价格高, 受益人将选择从让人购买资产, 并在市场上将其卖出, 这将产生一个名为回报的利润。否则, 受益人不会交易期权, 但让人获得期权费。有一个期权的种类称为亚式期权, 它的特征是回报是基于资产的平均价格 (算术或几何平均) 而计算的, 并在预先安排的实例上进行采样。这减少了市场波动和短期市场操纵所带来的风险。

为了进行风险分析以及对亚式期权定价, 可以使用蒙特卡罗 (MC) 模拟方法。在该方法中, 基于可用的资产波动信息对资产价格的多个随机历史进行模拟。

变量 $S(t)$ 是期权的潜在资产价格, 并假定价格按随机方程随时间变化:

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t)$$

在此公式中, μ 是资产的偏差, σ 是期权的波动率, $B(t)$ 表示一个标准的布朗运动。这种随机微分方程的解可以写成:

$$S(t_i) = S(t_{i-1})e^{\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\chi}$$

其中, χ 是具有零均值以及单位标准偏差的正态分布随机变量, 并且 $\Delta t = t_i - t_{i-1}$ 。

为了计算该资产的亚式期权回报, 资产价格是在到期时间 T 内 N 个时刻下的平均价格:

$$\langle S \rangle_{\text{arithm}} = \frac{1}{N} \sum_{i=0}^{N-1} S(t_i)$$

$$\langle S \rangle_{\text{geom}} = \exp\left(\frac{1}{N} \sum_{i=0}^{N-1} \log S(t_i)\right)$$

以上分别为算术均值和几何均值, 其中 $t_i = T \times i / (N - 1)$ 。执行价格 K 的看涨期权以及看跌期权所对应的回报是:

$$P_{\text{put}} = e^{-rT} \max\{0; K - \langle S \rangle\}$$

$$P_{\text{call}} = e^{-rT} \max\{0; \langle S \rangle - K\}$$

其中，根据合同， $\langle S \rangle$ 可以是 $\langle S_{\text{arithm}} \rangle$ 或 $\langle S_{\text{geom}} \rangle$ ， r 为无风险利率。

在特定参数组合 $\{S, K, R, \mu, V, T, N\}$ 以及平均规则（算术平均或几何平均）下，可以采用 MC 模拟从数值上决定亚式期权的数学期望。该模拟可以发出 M 个（大量的）随机路径，其中每条路径在 $t = 0$ 以及 $t = T$ 间根据上述随机方程的解随机产生期权价格，并计算 N 个时间点的均值。这些均值随后用来计算看涨或看跌期权回报。最后，对这些在 M 条随机路径上的回报求均值，从而得到 MC 估计的数学期望。

25.2 应用设计

MC 方法是一种对选定系统的状态空间进行抽样的重要工具。它是一种易并行问题，因为可以在大量计算单元上同时产生许多独立的随机进程，且只需要在计算结束执行一个归约操作以收集那些独立路径的分布。此外，每个生成路径的资产价格是在过期时间 T 内 N 个时刻的均值，如前一节所示。因此，相同的算术运算可以应用到许多同时随机产生的路径，因此可以利用单指令多数据（SIMD）的思想来完成该计算。因此，用于计算亚期权定价分布的 MC 方法很容易在多个层次进行并行化，包括利用向量处理单元（VPU）、多线程，甚至利用集群环境下的多台机器（见图 25-1）。

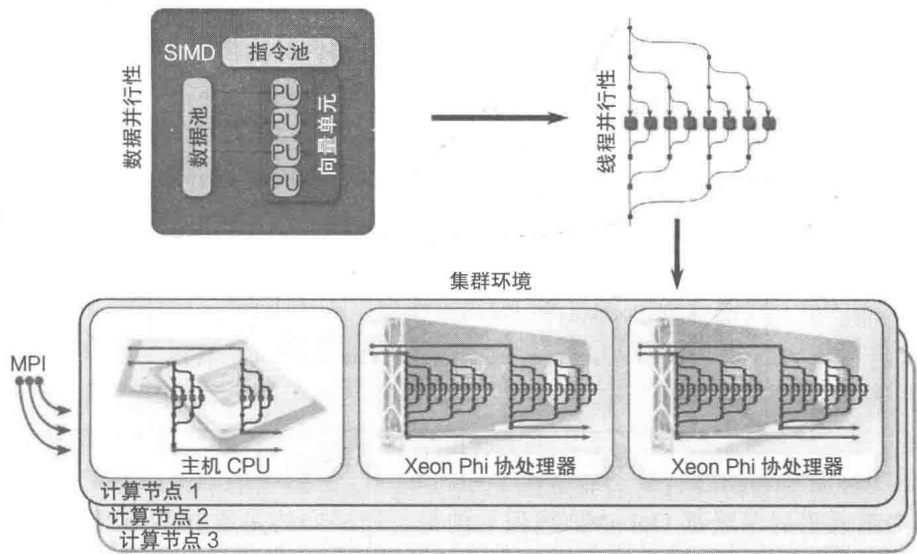


图 25-1 三个层次的并行

Intel 集成众核（MIC）架构将众多的 Intel 处理器内核集成到一个芯片上。每个 Intel Xeon Phi 7XXX 协处理器包含 61 个物理内核，每个内核拥有 4 个硬件线程和一个 VPU。独立的并行数据可由 VPU 上的 SIMD 指令处理（图 25-1；数据并行）。对于每个 Intel Xeon Phi 协处理器，并行的任务 / 指令可以分配在 244 个（61 核 × 4 个硬件线程）逻辑线程（图 25-1；线程并行）上。Intel Xeon 处理器上的 AVX 指令集拥有 256 位的向量寄存器。对于 Intel Xeon Phi 协处理器，向量寄存器的大小是 512 位。这可以将多达 16 个单精度浮点数或 32 位短整数，以及 8 个双精度浮点数或 64 位长整数打包到一个向量寄存器中，并对其同时进行

操作。对于计算密集型应用，多个协处理器以及主处理器均可用于计算。而计算集群中设备和节点之间的通信可以通过 MPI 实现。

为了实现高性能的亚式期权定价计算，应用程序代码设计需要更好地利用数据和线程并行性。这将会使用 Intel Xeon Phi 上的宽向量寄存器，并且也会将这些并行计算划分到众多内核上。这可以通过两个嵌套的 for 循环实现。外层循环通过线程并行库扩展到多个内核上，而内层循环通过 Intel C/C++ 编译器实现自动向量化，其中自动向量化是在编译器 -O2（默认的）优化选项中所开启的一种优化技术。

图 25-2 中的源代码显示了外层 i 循环上的 OpenMP parallel for 编译制导语句，它将循环迭代分布到由 OpenMP 初始化的所有可用线程上。内层的三个 k 循环各自的迭代次数为 vecSize，并与向量寄存器大小成比例，这将被编译器自动向量化。随机数生成器的 Intel Math Kernel Library 实现也采用了向量指令。

```
/* The i-loop is thread-parallel, i.e.,
distributed across the processor cores */
#pragma omp parallel for schedule(guided) \
reduction(+: payoff_arithm_put)
for (int i = 0; i < nPaths/vecSize; i++) {
    for (int j = 1; j < nIntervals; j++) {
        /* Intel MKL random number generator */
        vsRngGaussian(
            VSL_RNG_METHOD_GAUSSIAN_BOXMULLER,
            stream, vec_size, rand, 0.0f, 1.0f);
        /* The k-loop is data-parallel thanks to
        automatic vectorization by the compiler */
        for (int k = 0; k < vecSize; k++) {
            spot_prices[k] *=
                exp2f(drift + vol*rand[k]);
            sumsm[k] += spot_prices[k];
        }
        for (int k = 0; k < vecSize; k++) {
            arithm_mean_put[k] =
                K - (sumsm[k] * recipIntervals);
            if (arithm_mean_put[k] < 0.0f)
                arithm_mean_put[k] = 0.0f;
        }
        /* Reduction across vector lanes and across
        OpenMP threads is automatically implemented
        by the compiler */
        for (int k = 0; k < vecSize; k++)
            payoff_arithm_put += arithm_mean_put[k] *
                expf(-r*T)/(float)nPaths;
    }
}
```

图 25-2 亚式期权定价计算的计算密集部分的代码

多个向量通道以及全部 OpenMP 线程上的最终结果归约是通过编译器以及 OpenMP 库中的归约操作自动执行的。因此，基于数据并行的亚式期权回报的 MC 计算可以扩展到众多内核和 VPU 上，且同时对多个独立的向量通道进行操作。

下一节将讨论如何通过将计算参数（执行价格）集分发到多个计算单元（处理器以及协处理器）上实现另一个层次的并行化。

25.3 异构集群中的同步

在 MPI 通信模型中，多台机器和设备可以是分布内存架构，并将它们组合起来运行一个并行 MPI 应用程序，在集群环境中将应用程序尽可能地在可用的计算设备上扩展。这个高层次的 MPI 并行需要对多个设备上多进程之间的通信进行管理，但不提供任何自动

调度程序或负载均衡器，而是由程序开发人员负责。然而，在拥有不同计算性能的计算设备的异构集群环境中，工作负载均衡是一个十分重要的问题。

在最初设计的应用程序（参见图 25-3）中，100 个执行价格（nStrikes）被平均分配到所有 MPI 序列中。这些计算执行 nCalculations=10 次，并在每次计算步骤后有一个显式的 MPI_Barrier 进行同步。需要注意的是，Intel Xeon Phi 协处理器生成和计算对应的亚式期权定价价值的速度比两个 Intel Xeon 处理器要快，这导致协处理器空转且效率只有 79%，如图 25-5 的应用程序输出所示。

```
for (int iCalc=0; iCalc < nCalculations; iCalc++) {
    const int nStrikes = 100;
    const float strikeMin = 10.0f
    const float strikeMax = 20.0f
    for (int iStrike = myRank*nStrikes/(float)mpiWSize;
         iStrike < (myRank+1)* nStrikes/(float)mpiWSize;
         iStrike++) {
        const float K = strikeMin + (strikeMax - strikeMin)*
            (float)iStrike / (float)(nStrikes - 1);
        ...
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

图 25-3 非均衡的工作负载分配的代码清单

下一节将集中讨论 MPI 应用程序的两个优化方面。

- 不均匀的工作负载分配（或负载不均衡）导致一些 MPI 序列空转。
- 查找通信瓶颈，即延迟执行的代码段。

为实现上述优化技术，我们将使用 Intel 跟踪分析器与采集器（ITAC），以及一个 MPI 通信的可视化和性能分析工具（它是 Intel Cluster Studio 企业版的一部分）。ITAC 提供了一种方便的方法来测试和分析 MPI 通信并找到可提升性能的地方。

25.4 通过 ITAC 寻找性能瓶颈

ITAC 的两个构件——Intel 跟踪采集器（ITC）和 Intel 跟踪分析器（ITA）——会用来收集和分析 MPI 代码示例的跟踪数据。ITC 库的输出结构包括 MPI 参数记录以及所有函数调用以及 MPI 通信（包括用于同步等待的时间）的时间记录。这些 MPI 应用程序的空转（例如，等待同步所花费的时间）是性能提升的第一候选。

在用户的 MPI 应用程序中使用 ITC 库的一个简单方法是，在运行时加载 ITC 库并指定一个额外的 -trace 标记作为 mpirun 脚本执行的参数。ITC 库默认为每个 MPI 进程单独生成一个跟踪文件，但是输出文件的格式可由环境变量控制。例如，多个跟踪输出可通过 ITC 组合成一个包含所有 MPI 应用进程执行信息的跟踪文件。这个（或者这些）文件在用户应用程序运行结束时由 ITC 生成，之后可以在 ITA 中进行分析。

25.5 建立 ITAC

一个异构集群环境中可以包含具有不同带宽和计算能力的处理器和设备。对称的 MPI 应用程序会将相同的工作负载分配到应用程序的参与者中，这可能会导致负载不均衡，这是因为在具有较高计算性能的设备上，程序的执行时间可能会较短。MPI 同步事件会使负载不均衡显现出来，这是因为高性能的设备在等待较慢设备时会闲置。ITAC 可以将 MPI 通

信可视化。当一些 MPI 序列需要等待大量时间来与那些异构集群中执行速度较慢的设备进行同步时，ITAC 可以指出这些情况。

收集跟踪信息前，用户需要用以下脚本为 MPI 以及加载 ITC 库配置环境：

```
source /opt/intel/impi/4.1.3.048/intel64/mpivars.sh
source /opt/intel/itac/8.1.4.0.45/intel64/itacvars.sh
```

ITC 采用结构化跟踪文件（STF）格式来收集每个 MPI 进程的 MPI 通信时间记录。为了方便起见，可以通过设置以下环境变量使 ITC 将所有数据合并到一个跟踪文件中：

```
export VT_LOGFILE_FORMAT=singlestf
```

此时，用户的 MPI 应用程序已准备就绪。之后会将跟踪数据收集到单个文件中。

启动对 MIC 架构的支持需要使用三个 Intel MPI 环境变量，将 TCP/IP 设置为通信协议，并根据对应的 OpenMP 绑定策略将 MPI 进程进行绑定：

```
export I_MPI_MIC=1
export I_MPI_FABRICS=tcp
export I_MPI_PIN_DOMAIN=omp
```

下面的执行命令将在 machines-HETEROGENEOUS 文件所指定的设备上运行 MC 亚式期权定价计算，其中每个 MPI 进程运行在一个设备上。由于使用了 -trace 标记，因此会生成一个包含该应用所调用的全部 MPI 函数的时间信息的跟踪文件。

```
mpirun -trace -machinefile machines-HETEROGENEOUS \
-env I_MPI_PIN=0 ~/options &> results.txt
```

machine-HETEROGENEOUS 文件的内容包括主要节点的主机名以及相应的 Intel Xeon Phi 协处理器的主机名：

```
c001-n003
c001-n003-mic0
c001-n003-mic1
```

最后，ITA 可以用来打开生成的单个跟踪文件并开始分析：

```
traceanalyzer options.single.stf
```

在下一节中，我们将看到通过 ITAC 对 MPI 通信进行代码优化以及可视化的例子。

25.6 非均衡的 MPI 运行

图 25-4 显示了 ITA 默认的图形用户界面视图。它包含跟踪的时间间隔、MPI 函数调用的时间表，以及全部 MPI 进程的总执行 / 通信时间信息。

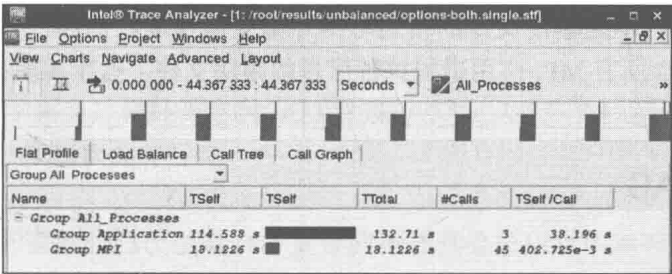


图 25-4 Intel 跟踪分析器：默认的图形用户界面视图

图 25-4 中 “Group Application” 表示全部 MPI 进程执行时间的总和。为了方便讨论，

我们将认为这是花费在有用计算上的时间。“Group MPI”收集所有 MPI 序列在通信以及同步方面所导致的空闲时间间隔。因此，为了达到更好的整体性能，该数量应尽可能小。

这种特殊的 MPI 运行使用了一个包含两插槽 8 核 Intel Xeon E5-2687W v2 处理器的计算节点（主机名 c001-n003），开启了超线程，且两个处理器同时由 MPI 的 rank 0 进程使用。内存总量为 128 GB，其中包含 8 个 1600 MHz 的 16GB 的模块。系统中安装了 4 个拥有 16GB GDDR5 内存的 Intel Xeon Phi C0PRQ-7120 被动冷却协处理器，但只有前两个用于计算，且分别运行 MPI 的 rank 1 以及 rank 2 进程。操作系统是 CentOS 6.5，Linux 内核为 2.6.32-431.el6.x86_64。MPSS 版本是 3.2.3，MPI 版本是 4.1.3.048，ITAC 版本是 8.1.4.0.45。

从 Flat Profile 切换到 Load Balance 可以显示每个 MPI 进程的执行时间，以及每个进程花费在 MPI 通信以及同步上的时间（见图 25-5）。

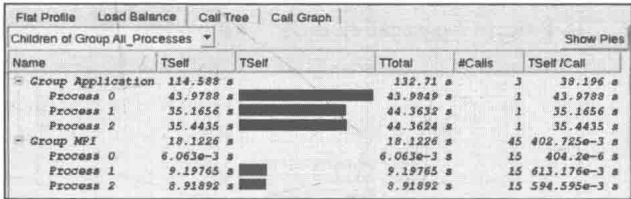


图 25-5 ITAC：负载均衡视图。三个执行亚式期权定价的 MPI 进程分别运行在主机以及两个 Intel Xeon Phi 协处理器上

总执行时间（显示在 TTotal 列）中的 44.3s 用在进程 1 以及进程 2 运行的协处理器上，且只有 35.3s 花费在每个进程的有用计算上，平均有 9.1s 用在 MPI 通信以及等待上（分别对应于 Group Application 以及 Group MPI 的 Tself 列）。

在应用程序的输出（见图 25-6）中可以发现同样的比例。第一列包含 MPI 进程所运行设备的主机名。Performance 列表示设备每秒产生的随机路径数。最后的 Effic. 列显示有用计算时间与负载不均衡导致的 MPI 通信空闲时间，以及三种工作设备之间的同步时间的比值。Net performance 是全部路径与全部计算时间（包含进程的空闲时间）的比值。因此，网络性能总是比单个设备性能的总和小，并且只有在完美的负载均衡下才会相等。

#	Worker	Share	Performance	Effic.
	c001-n003	34.0%	8.19e+06	100.0%
	c001-n003-mic0	33.0%	1.06e+07	77.3%
	c001-n003-mic1	33.0%	1.05e+07	78.1%
#	Calculation	7 of 10	took 4.356 seconds	
#	Net performance:		2.41e+07 paths/second	

图 25-6 不均衡的亚式期权定价 MPI 运行的输出结果片段

默认情况下，ITAC 使用 MPI 进程编程（例如，Process 2）来显示执行时间的统计数据。可以更改组名称为主机名，方法是单击菜单 Advanced → Process Aggregation，从列表中选择 AuNodes，然后单击 Application 或 OK 按钮（见图 25-7）。

以上操作的结果如图 25-8 所示，在主机 c001-n003-mic1 上执行时间接近 44s，而 Intel Xeon Phi 协处理器 c001-n003-mic0 和 c001-n003-mic1 上有用计算时间接近 35s，其净效率约为 80%。

使用菜单 Charts → Event Timeline 或者键盘快捷键 Ctrl+Alt+E，可以对非均衡 MPI 亚式期权定价计算中设备之间的通信进行可视化。由于之前修改了所有主机的进程聚合方法，主机名组也会显示在这里。

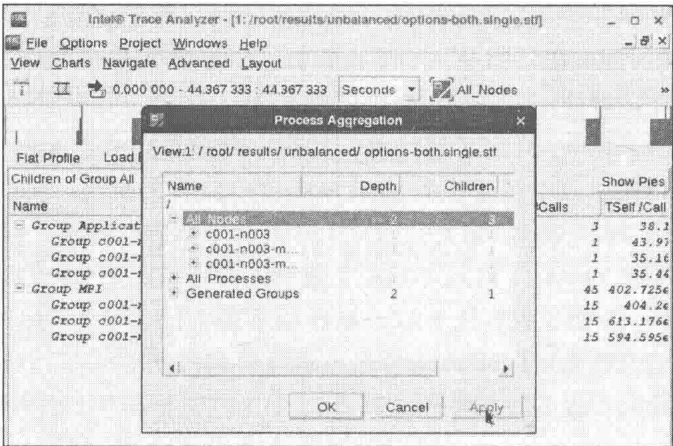


图 25-7 ITAC: 将 Process Aggregation 改为 “All Nodes” 可以看到设备的主机名

Flat Profile Load Balance Call Tree Call Graph					
Children of Group All_Nodes					
Name	TSelf	TSelf	TTotal	#Calls	TSelf / Call
Group Application	114.588 s		132.71 s	3	38.196 s
Group c001-n003	43.9788 s		43.9788 s	1	43.9788 s
Group c001-n003-mic0	35.1656 s		44.3632 s	1	35.1656 s
Group c001-n003-mic1	35.4435 s		44.3624 s	1	35.4435 s
Group MPI	18.1226 s		18.1224 s	45	402.725e-3 s
Group c001-n003	6.063e-3 s		6.063e-3 s	15	404.2e-6 s
Group c001-n003-mic0	9.19765 s		9.19765 s	15	613.176e-3 s
Group c001-n003-mic1	8.91892 s		8.91892 s	15	594.595e-3 s

图 25-8 ITAC: Load Balance 选项卡，显示按设备的主机名分组的执行时间

Event Timeline 图显示了每个序列或设备上应用程序执行时间的统计分析 (参见图 25-9)。导航键可以用于放大 / 缩小，以及移动所选择的时间间隔。请注意，缩放或平移区间时，除了顶部的 Trace Map 外，所有图将会变为只显示所选时间段的信息。

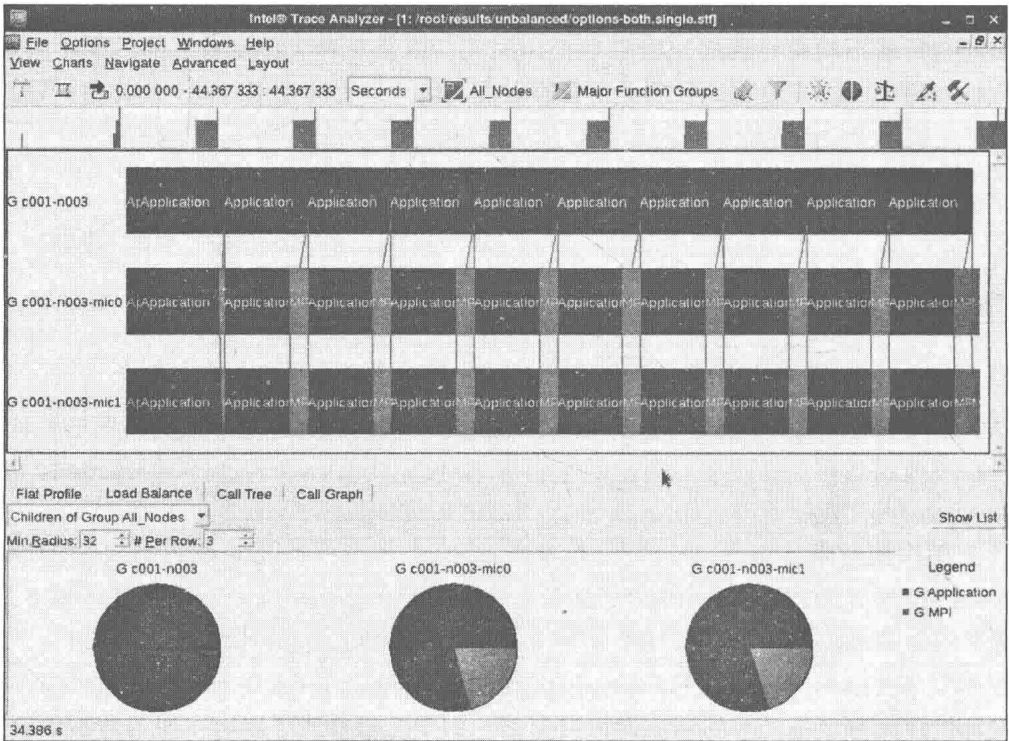


图 25-9 ITAC: 非均衡 MPI 运行的事件时间轴

ITAC 中所有图使用以下着色方案：标有“Application”的蓝色（书中为深灰色）栏代表有用的计算，标有“MPI”的红色（书中为浅灰色）栏表示 MPI 通信以及空闲时间；点对点通信用黑线与对应的 MPI 组相连，并由蓝色线表示集合通信，但在蓝色书中的“Application”栏背景下不容易被发现。因此，可以在时间轴图上右击并选择 Event Timeline Settings 菜单，之后修改对应的参数，以实现颜色的修改。

25.7 手动负载均衡

一种将示例应用的计算工作负载进行分配的方法是将更多 iStrike for 循环的迭代分配到更快的设备上。当所有迭代花费相同的处理时间时，以下方法是可行的，这也正是这个特殊问题所面临的情况。如果执行时间对于每次迭代是不同的，例如由于非确定性，这种方法仍然可能导致不均衡的工作负载分配（图 25-10）。

```
int arrStrikes[4] = {0, 28, 64, 100};
for (int iCalc=0; iCalc < nCalculations; iCalc++) {
    const int nStrikes = 100;
    const float strikeMin = 10.0f
    const float strikeMax = 20.0f
    for (int iStrike = arrStrike[myRank];
        iStrike < arrStrike[myRank+1];
        iStrike++) {
        const float K = strikeMin + (strikeMax - strikeMin)*
            (float)iStrike / (float)(nStrikes - 1);
        ...
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

图 25-10 手动实现均衡的工作负载分配的代码清单

迭代次数是基于应用程序输出的性能值而计算得出的（见图 25-6）。MPI 运行时将采用 Intel Xeon 处理器作为 rank0，两个协处理器分别作为 rank1 和 2。因此，前 28 次迭代将分配给处理器，剩余的两组 36 次迭代（共计 100 次迭代）分别分配给两个协处理器：

$$N_{CPU} \approx \frac{8.19 \times 10^6}{8.19 \times 10^6 + 1.06 \times 10^7 + 1.05 \times 10^7} \times 100 \approx 28$$
$$N_{PHI} \approx \frac{1.05 \times 10^7}{8.19 \times 10^6 + 1.06 \times 10^7 + 1.05 \times 10^7} \times 100 \approx 36$$

这些迭代次数作为数组 arrStrikes [4] 的值存储在应用程序中，并分配给计算的参与者。例如，第一个协处理器将处理第 28 ~ 64（64 = 28 + 36）次执行，共处理 36 次执行迭代。因此，对计算设置的任何修改也将需要重新计算这些值并对代码进行重新调优。

用于手动均衡计算的 Event Timeline 图显示在图 25-11 的底部。由于更少的时间用在 MPI 进程之间的同步上，因此提高了应用程序的整体性能。

单击 View → Compare 可以将新的跟踪与原始的不均衡情况进行对比。可以同步导航键或鼠标缩放，以及使两个跟踪的时间尺度相一致。在这里你可以看到一些改进，应用程序在实际计算（蓝色区域；书中为深灰色）上正在花费更多的时间，而不是等待通信（见图 25-12）。

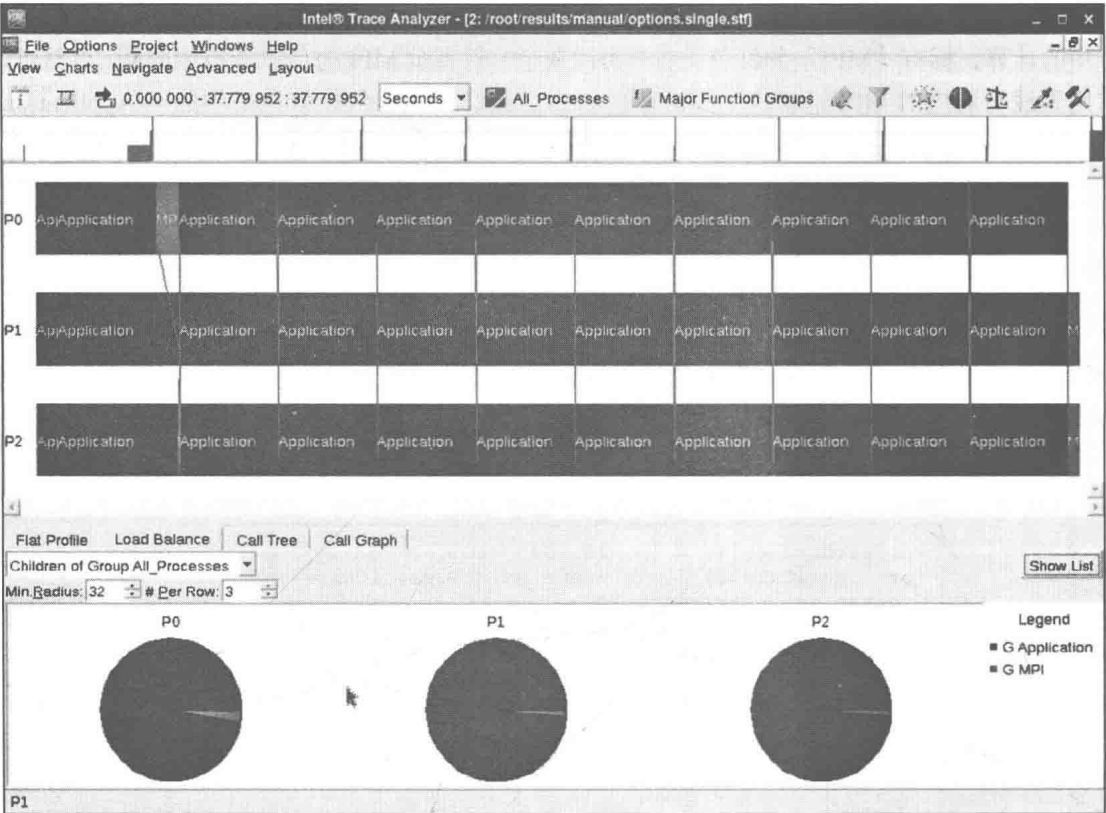


图 25-11 ITAC：手动均衡 MPI 运行的事件时间轴

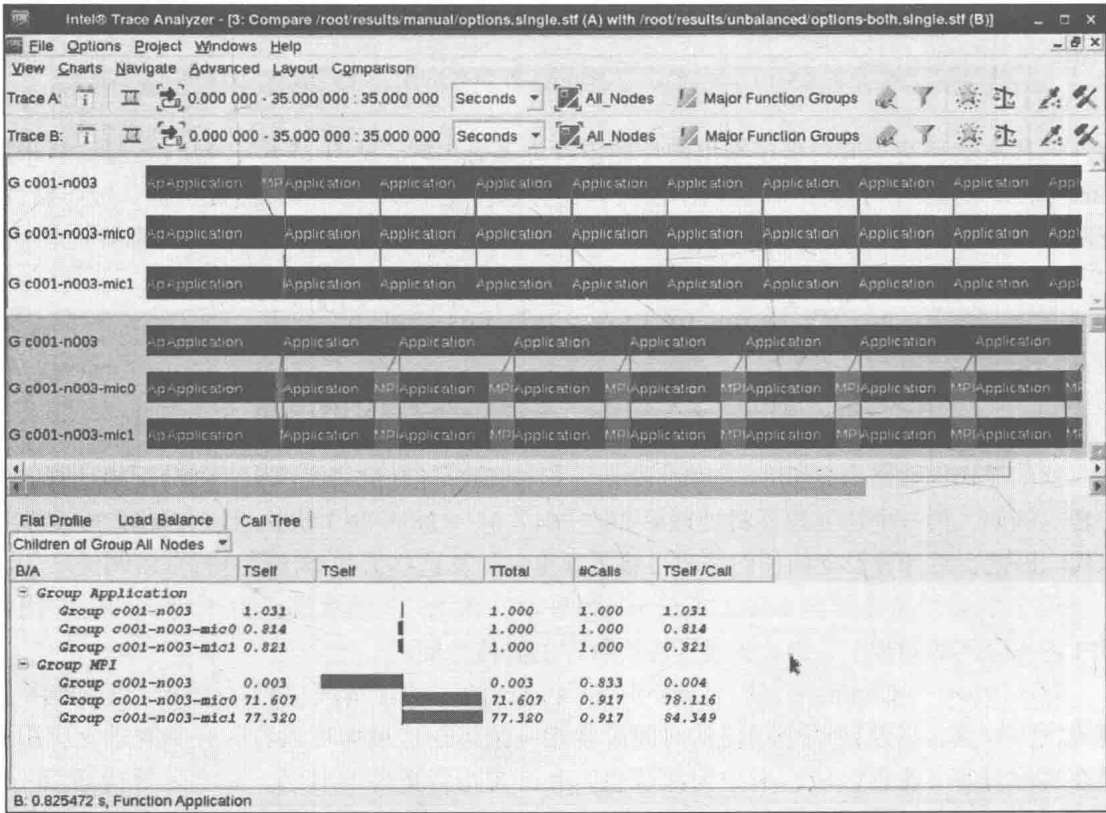


图 25-12 ITAC：手动均衡和原来非均衡 MPI 运行的事件时间轴比较视图

手动均衡的 MPI 应用程序运行输出结果表明，该实现相对于非均衡的代码运行速度提高了 22%（见图 25-13）。

#	Worker	Share	Performance	Effic.
	c001-n003	28.0%	8.26e+06	99.2%
	c001-n003-mic0	36.0%	1.06e+07	99.5%
	c001-n003-mic1	36.0%	1.05e+07	100.0%
#	Calculation	4 of 10	took 3.587 seconds	
#	Net performance:	2.92e+07 paths/second		

图 25-13 手动均衡的亚式期权定价 MPI 应用程序运行的输出结果片段

25.8 动态老板 – 工人负载均衡

负载均衡通常需要改变整个 MPI 应用程序上的工作负载分配。本节讨论在异构集群组件上的动态负载分配（见图 25-14）。

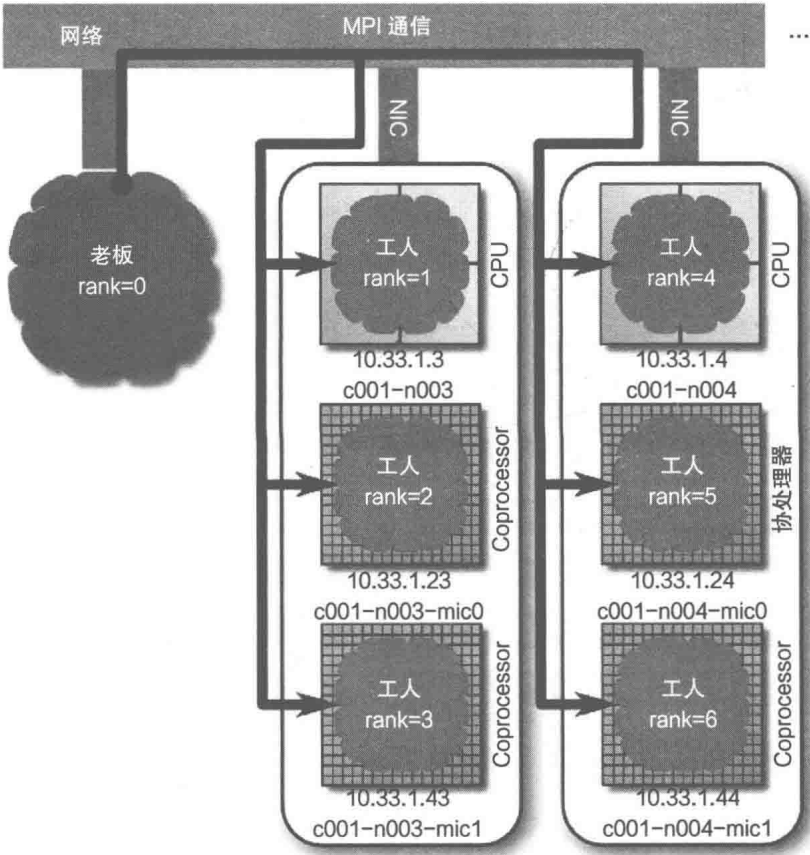


图 25-14 动态负载均衡的老板 – 工人模型下的 MPI 通信方案

指定 MPI rank0（老板序列）负责动态调度，并且在异构集群组件（工人）中分配 MC 亚式期权定价计算，这里的异构集群组件为 Intel Xeon 处理器以及 Intel Xeon Phi 协处理器。在之前的计算步结果返回时，每个工人请求一个新的计算部分。其结果是，在计算集群中更快的组件比慢组件执行更多的计算。在此期间无需手动干预，只需一个线程专门负责工作负

载分配 (图 25-15 和图 25-16 中的 MPI process0)。

应当指出的是, 如果每次所请求的工作负载量是非常小的, 工人和老板之间的通信可能成为瓶颈。这可以通过增加每次分配给每个工人的计算量来解决。另一方面, 如果工作负载分布的粒度过粗, 这可能会导致最后请求的不均衡处理。因此, 我们努力在请求的通信以及每次请求所分配的工作量之间寻找均衡。

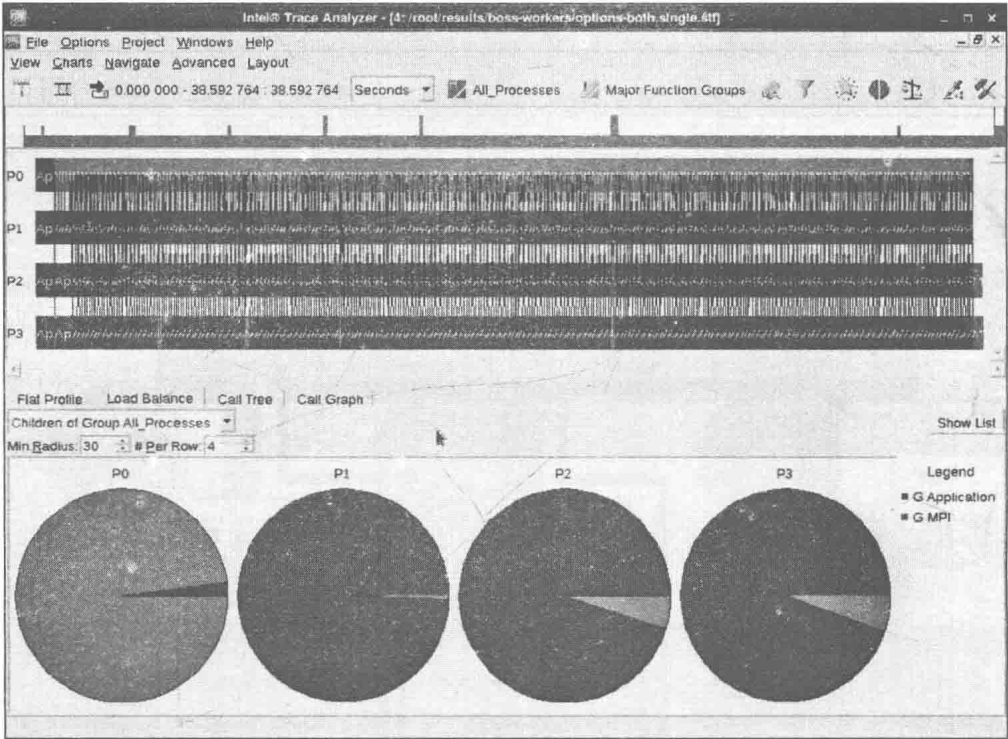


图 25-15 ITAC: 动态均衡的老板 – 工人模型实现的事件时间轴

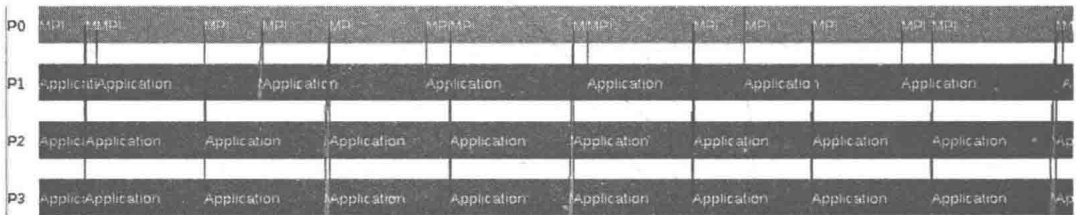


图 25-16 ITAC: 动态均衡的老板 – 工人模型实现的放大的事件时间轴

图 25-16 显示了放大之后的老板进程 0 和三名工人进程 P1、P2 和 P3 之间的通信, 其中最后两个进程运行在 Intel Xeon Phi 协处理器上。

应当指出的是, 老板进程只需要一个线程。剩余的可用线程可以作为工人使用。但是在默认情况下, 当在同一设备上运行两个 MPI 进程时, MPI 会将所有可用的内核平均分为两组, 每组提供设备 50% 的计算能力。通过使用下面两个环境变量, 计算性能会根据每个 MPI 进程上所使用的 OpenMP 线程数而按比例分配:

```
export I_MPI_PIN_DOMAIN=omp
export I_MPI_PIN=0
```


由老板序列实现动态工作负载分布的代码段如图 25-17 所示。
动态均衡的老板 - 工人实现所产生的执行结果与图 25-18 给出的结果类似。
图 25-19 显示了本章所讨论的三种实现的整体性能，并由每秒所产生的随机路径数表示。

```
if (myRank == bossRank) {
    int nR = 0; /* Number of processed tasks */
    int iP = 0; /* Next task to assign */
    while (nR < nPars) {

        /* Wait for any worker to report for work */
        float buf[msgReportLength];
        MPI_Recv(&buf, msgReportLength,
            MPI_INT, MPI_ANY_SOURCE, msgReportTag,
            MPI_COMM_WORLD, &status);
        const int iW = status.MPI_SOURCE;

        if (buf[0] > 0.0f) {
            /* If worker reports with results of a
            previous task, record these results */
            nR++;
            const int iR = floorf(buf[1]);
            payoff_arithm_put [iR] = buf[2];
        }

        if (iP < nStrikes) {
            /* Assign the next task iP to worker iW */
            float buf[msgSchedLen] = {iP,
                M[iP], N[iP], K[iP], S[iP], /*...*/};
            MPI_Send((void*)&buf, msgSchedLen,
                MPI_FLOAT, iW, msgSchedTag,
                MPI_COMM_WORLD);
            iP++;
        }
    }
}
```

图 25-17 动态均衡的老板 - 工人模式下亚式期权定价 MPI 应用程序的源代码片段（老板序列）

#	Worker	Share	Performance	Effic.
	c001-n003	28.0%	8.10e+06	99.5%
	c001-n003-mic0	36.0%	1.07e+07	96.9%
	c001-n003-mic1	36.0%	1.07e+07	96.6%
# Calculation 9 of 10 took 3.645 seconds				
# Net performance: 2.88e+07 paths/second				

图 25-18 动态均衡的老板 - 工人模式下亚式期权定价 MPI 应用程序运行的输出片段

图 25-19 中的性能结果表明，对于亚式期权回报的 MC 计算，老板 - 工人动态负载均衡与静态 / 手动负载均衡具有类似的性能。

25.9 结论

与 GPU 不同，Intel Xeon Phi 协处理器在与主处理器对称的模式下执行本机应用。在这种模式下，应用程序在协处理器的操作系统上运行，并且不需要运行在 CPU 上的主机进程卸载数据到协处理器。因此，对于一个 MPI 框架中的应用程序，可以在协处理器上直接运行 MPI 进程，并与主机系统处理

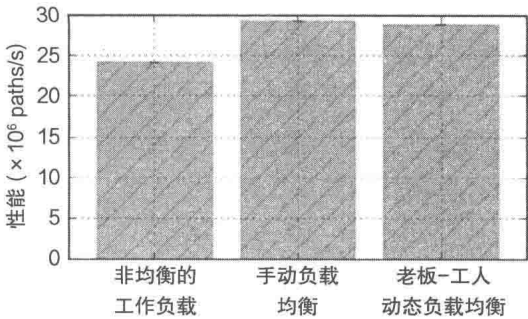


图 25-19 本章提出的三种负载均衡技术的性能结果

器运行相同的代码（对称模式）。在这种情况下，与集群中的独立计算节点一样，协处理器拥有一个 MPI 序列以及端到端通信能力，并且可以访问网络共享文件系统。在这样的配置下，就没有必要测量应用程序中的数据卸载以达到异构系统中处理器和协处理器的折中使用。这就是说，在无需修改的前提下，一个专门为 CPU 集群设计的 MPI 应用程序也可以用于拥有协处理器的集群环境中。

为了使对称的 MPI 应用程序获得更好的整体性能，工作负载均衡是必需的，因为异构集群中的计算单元（处理器和 Intel Xeon Phi 协处理器）拥有不同的计算能力。

显式或手动负载均衡可以通过根据计算设备的计算能力成比例重新分配计算工作量来实现。

动态负载均衡技术（老板 - 工人模式）可以产生与手动工作负载分布类似的整体性能。这种方法有另一个优点：MPI 应用程序的扩展不需要额外的程序调优。通过添加更多计算节点以及在机器文件中写入更多主机名，基于动态负载分布实现的 MPI 应用程序即可获得更快的计算速度。

在此提出的查找和解决对称 MPI 应用程序中负载不均衡的方法，可以用于提高异构集群环境（主处理器以及 Intel Xeon Phi 协处理器）中应用程序的整体性能。这种集群上的统一编程环境使我们的工作非常简单。所生成的代码具有广泛的适用性，它同样适用于每个节点之间具有执行差异的同构系统。

25.10 更多信息

与本章相关的一些额外的阅读材料如下：

- Vladimirov, A., Karpusenko, V., October, 2013. Heterogeneous Clustering with Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors. Colfax Research. <http://research.colfaxinternational.com/post/2013/10/17/Heterogeneous-Clustering.aspx>.
- Colfax International, 2013. Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. ISBN-10 0-9885234-1-8|ISBN-13 978-0-9885234-1-8. <http://www.colfax-intl.com/nd/xeonphi/book.aspx>.
- 本章以及其他章的代码下载地址 <http://lotsofcores.com>。

集群上可扩展 OOC 解法器

Eduardo D'Azevedo*, Ki Sing Chan†, Shi-Quan Su‡, Kwai Wong‡

* 美国, 橡树岭国家实验室; † 中国, 香港中文大学; ‡ 美国, 田纳西大学

本章介绍配备 Intel Xeon Phi 协处理器的集群系统上分布式核外 (out-of-core, OOC) 大规模稠密矩阵 LU 和 Cholesky 分解解法器的实现。OOO 算法融合旨在减少 CPU 和协处理器间数据移动的 left-looking 和 right-looking 技术, 同时优化数据局部性和计算吞吐量。OOO 解法器的接口与 ScaLAPACK 软件库一致, 因此可以方便地用于调用 ScaLAPACK 的已有程序。实验性能测试将在配备 48 个节点的美国国家计算科学研究所 Beacon 集群系统上进行, 其中每个节点都配置 Intel Xeon 处理器以及 Intel Xeon Phi 协处理器, 同时还将给出 Intel Xeon Phi 协处理器以及 GPU 集群上的性能对比。

26.1 引言

在现代数值模拟 (例如热分析、边界元法分析、混合应用中的电磁波计算) 中, 大规模稠密矩阵计算均是其骨干函数。许多顶级的超级计算机都利用配置 PCIe 的特殊加速器或类似 Intel Xeon Phi 的协处理器, 或者 NVIDIA 图形处理器。例如, 2014 年 6 月发布的 TOP500 中的第一名是位于中国广州国家超级计算中心的天河 2 号超级计算机, 它利用 Intel Xeon Phi 协处理器计算 LINPACK 达到峰值性能 33.86PFLOPS, 排名第二的是位于美国橡树岭国家实验室的 Cray XK2 Titan, 它使用 NVIDIA K20 GPU 计算 LINPACK 性能达到 17.59PFLOPS。

尽管协处理器和 GPU 加速器具有更高的数值计算能力, 但是在高效利用这些辅助设备方面依然存在一些挑战:

- 为了有效利用加速器或协处理器, 需要开发大规模并行性。Intel Xeon Phi 的协处理器有 60 个物理 x86 处理内核, 每个处理器内核支持 4 路多线程处理, 因此总体可支持 240 路并行性。
- 与主计算机内存相比, 辅助设备仅有有限的设备存储器, 例如, 协处理器可能有 8GB 片上内存, 但是主机端可以配备 32GB 或更多主存。因此需要外存或 OOC 算法来解决需要更大设备存储器的问题。
- 主机或者主机内存与设备间的数据传输非常耗时, 例如 Cray XK7 Titan 超级计算机的 GPU 和 CPU 主机间的数据传输带宽是 4GB/s, 因此, 设备上后续的计算需要摊销费时的数据传输。

因此, 新的并行 OOC 解法器需要高效利用这种包括多核 CPU 和协处理器或加速器的新型混合体系结构。已经存在大量异构系统上的稠密线性代数库的实现, (参见 26.7 节, Agullo et al., 2011, 2009; Barrett et al., 2010; D'Azevedo and Hill, 2012; Fogue et al., 2010; Humphrey et al., 2010; Jetley et al., 2010; Quintana-Orti et al., 2009; Song and Dongarra, 2012; Song et al., 2012)。一般而言, 这些数值库函数通过大量优化和调优来达到较高吞吐量。ScaLAPACK

库中已经使用了与二维循环分块分配兼容的新的并行 LU 解法器，以充分利用了 NVIDIA GPU。该算法使用 OOC 方法，将 GPU 设备存储器视为快速内核存储器，而将主机端的主存视为慢速的辅助存储器。OOO 算法在分解阶段采用 left-looking 和 right-looking 算法，left-looking 算法最小化通信开销，用来更新协处理器上的矩阵，而 right-looking 算法可提供更高的计算性能。

本章介绍利用 NVIDIA GPU 上 CUBLAS 以及 Intel Xeon Phi 协处理器卸载模式和 Intel MKL 来使用并行 OOC 解法器中的技术和挑战。首先，26.2 节展示在多核和多 GPU 系统上以及多核和多协处理器系统上并行分解大规模稠密矩阵的 OOC 算法。接下来，26.3 节介绍移植 GPU 解法器到协处理器上的技术和挑战。26.4 节将给出 Beacon 上的数值结果，包括性能数据和 Keeneland 上的时间比较结果。最后，26.5 节给出了本章的精华和一些正在进行的工作。

26.2 基于 ScaLAPACK 的 OOC 分解算法

在过去 CPU 内存相比外存大小有限时，大规模稠密矩阵问题的 OOC 分解算法已被大量研究。其中的本质思想也可应用于目前的异构系统。本章中，我们利用 E. D'Azevedo 和 J. Dongarra（参见 26.7 节；D'Azevedo and Dongarra, 2000）提出的 left-looking OOC 算法来实现 LU 和 Cholesky 分解，并且介绍一个旨在减少 CPU 和辅助设备存储器间的数据传输的优化方法。OOO 算法的核心是一个核内并行 LU 和 Cholesky 分解，即仅利用 right-looking 算法计算设备上数据的算法。

ScaLAPACK 使用分布式的二维递归分块存储格式，如图 26-1 所示。主机端使用基于 MPI 的 BLACS（基本线性代数通信子程序）来实现数据传输，一些论文（参见 26.7 节；D'Azevedo and Dongarra, 2000；D'Azevedo and Hill, 2012）已经详细介绍了并行 LU 算法。本章中，我们将类似的 OOC 算法扩展到 Cholesky 分解中。

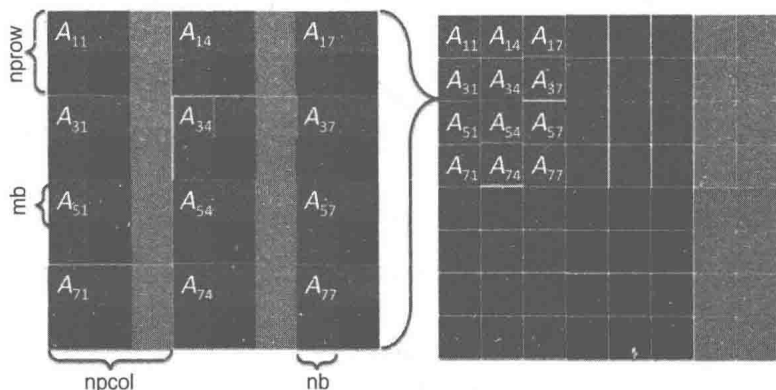


图 26-1 ScaLAPACK 中的二维分块循环分布。矩阵 A 被一个大小为 $\text{nprow} \times \text{npcol}$ 的处理器网格划分， nb 是块大小

26.2.1 核内分解

ScaLAPACK 的 PDPOTRF 函数使用 right-looking 算法计算 Cholesky 分解，对设备内存中的分布式矩阵也采用同样的方法。由于 CPU 主机不能直接访问辅助设备的存储，因此设

备上的数据需要传输到 CPU 上的一个临时缓冲区中,以便被 MPI 库访问。这种传输是主要的性能瓶颈,因此为了获得较高性能需要优化这一点。

考虑对称正定矩阵 A 的分块划分:

$$A = \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^t & L_{21}^t \\ & L_{22}^t \end{pmatrix} \quad (26-1)$$

其中,矩阵 A_{11} 是一个 $k \times k$ 的方阵。对于分解矩阵 A 需要进行一系列操作:

1. 假设前 k 列已经分解,应用:

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \rightarrow \begin{pmatrix} \tilde{A}_{11} \\ \tilde{A}_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} (L_{11}^t) \quad (26-2)$$

其中, $\tilde{A}_{11} = L_{11}L_{11}^t$, $\tilde{A}_{21} = L_{21}L_{11}^t$, 对角线矩阵块 A_{11} 的分解利用 ScaLAPACK 的 PDPOTRF 函数在 CPU 端执行,这里 $k = MB$ 为矩阵分块大小。

2. 在 GPU 端对 A_{22} 执行对称秩 k 更新。

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^t \quad (26-3)$$

这个任务是分解中的主要工作量,需要在处理器列中广播 L_{21} , 在处理器行中广播 L_{21}^t , 并在 GPU 端利用 DSYRK 和 DGEMM 执行计算,这里并不需要更多的通信。

3. 递归分解剩余矩阵

$$\tilde{A}_{22} = L_{22}L_{22}^t \quad (26-4)$$

right-looking 方法提供了更好的负载均衡并且提供更多的并行性。

26.2.2 OOC 分解

OOC 分解方法十分类似于上节中的核内算法。主要的不同之处在于,位于 CPU 端的矩阵 A 无法全部存储于设备端的核内存储器上,因此需要导致 CPU 和设备之间的一些数据传输,但是必须尽可能减少这些数据传输以获得较高性能。right-looking 方法和 left-looking 方法的总 I/O 开销分别是:

$$\frac{M^3}{3n_b} (1 + O(n_b / M))(R + W) \quad (26-5)$$

$$\frac{M^3}{2n_b} (1 + O(n_b / M))R + 2M^2 (1 + O(n_b / M))W \quad (26-6)$$

其中, n_b 是 $M \times M$ 大小矩阵 A 的分块大小, R 和 W 分别为读取和写入一个矩阵元素的开销。假定读写开销近似,则 left-looking 方法(即式(26-6)),较之 right-looking 方法(即式(26-5))而言具有更小的数据传输开销。

OOC 计算需要使用两个列块。大的列块 Y 位于设备存储中,并累加上一次分解中得到的更新,小的列块 X 保留上一次已计算完成的分解。计算的步骤基本类似核内算法。

1. 类似于核内分解,把 \tilde{A}_{12} 和 \tilde{A}_{22} 复制到设备内存的列块 Y 中,把已经计算的 L_{11} 和 L_{21} 中部分结果复制到设备内存的列块 X 中。

2. 将列块 Y 复制回 CPU 端然后调用 PBLAS 中的 PDTRSM 来进行三角矩阵求解。

3. 对列块 Y 进行对称秩 k 更新,即矩阵乘法。类似于核内分解,这需要沿处理器列广播

L_{21} 的部分元素, 并沿处理器行广播 L_{21}^t 的部分元素。然后可以调用 DSYRK 在对角线上分块上执行对称秩 k 更新, 并调用 DGEMM 处理其他分块, 这些操作都不需要通信。

4. 在所有的更新执行完毕后, 列块 (\tilde{A}_{22}) 的底部方形区域利用 26.2.1 节中介绍的核内算法进行分解。

5. 计算完成的列块 Y 被复制回 CPU 端设备中。

列块 Y 的宽度选择影响算法的性能, 并且该数值选择也受限于设备端的可用内存大小。较宽的列块 Y 会降低主机端和设备端的数据传输量。图 26-2 展示了 Cholesky 分解的 OOC 算法总体框架。

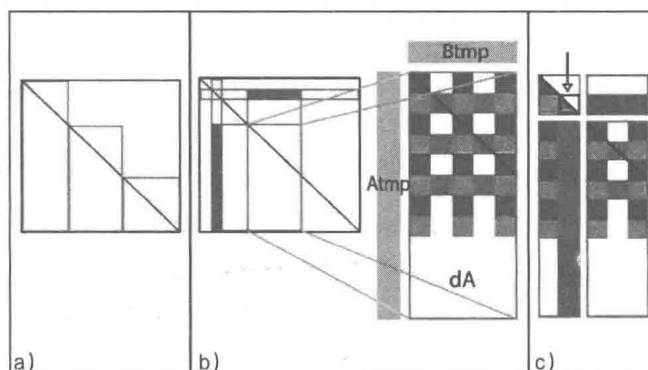


图 26-2 Cholesky 分解的 OOC 算法。a) 未计算的 A 矩阵被分解为列块 (左图)。b) 对于每一个列块使用 left-looking 更新算法。c) 使用 right-looking 算法

如果选择列块 Y 的宽度为 N/K , 即列块的总大小为 $N*(N/K)$, 或者说整个矩阵分为 K 个列块 Y 。第一个列块 Y 不需要任何预先计算的数值, 可以直接进行分解。但是接下来的 K 个 Y 列块分解都需要前 $K-1$ 列块 Y 的计算结果。总共的数据传输量为 $(1 + 2 + \dots + (K-1)) * (N*(N/K)) = (K-1)/2 * N^2$ 。因此 K 的值越小越好, 以及列块的宽度越大越好, 这样将极大地减少设备端和 CPU 端的数据传输量。然后, 宽度 N/K 受限于设备端的可用内存大小。

26.3 从 NVIDIA GPU 移植到 Intel Xeon Phi 协处理器

GPU 端的代码使用 CUDA 语言编写, 并利用 CUBLAS 中的高层库例程执行 BLAS 中的稠密矩阵操作, 进行数据传输管理以及设备端内存的分配和释放。这种利用高层操作的模式十分适用于协处理器上使用 Intel LEO (卸载语言扩展) 中 #pragma 编译制导语句的卸载模型。这种方法可以通过利用 OpenMP 4.0 中的 “target” 制导语句实现。

两者间的第一个不同之处在于设备内存的管理。卸载模式假定设备内存端的数组均有一个主机内存端的对应数组镜像副本, 使用图 26-3 中卸载指令给出的 length()、alloc_if()、free_if() 等选项执行操作。

```
double *Y = (double*) malloc(n*sizeof(double));
#pragma offload_transfer target(mic:MYDEVICE) nocopy(Y:length(n) alloc_if(1) free_if(0))
#pragma offload_transfer target(mic:MYDEVICE) nocopy(Y:length(n) alloc_if(0) free_if(1))
```

图 26-3 Intel MIC 上分配和释放内存的代码段

CUDA 允许显式的设备内存分配和释放，在主机端并不需要有对应的数据空间。图 26-4 中使用不同的长度将会导致错误。

```
int *p = (int*) malloc(1*sizeof(int));
int *q = (int*) malloc(1*sizeof(int));
#pragma offload_transfer target(mic) nocopy(p:length(1000) alloc_if(1) free_if(0))
#pragma offload_transfer target(mic) nocopy(q:length(1000) alloc_if(1) free_if(0))
```

图 26-4 不一致的内存卸载分配大小将会导致错误

一种典型的工作方式是在设备端利用 memalign() 分配内存，但利用长整型而非指针类型将地址指针传回，整数值只应用在卸载数据传输中，并且在使用前重新转换为指针类型。图 26-5 展示了 cublasAlloc() 和 cublasFree() 两个函数的等价实现。注意，intprt_t 类型是 C99 标准，保证可以存放一个指针数据。

```
intprt_t offload_Alloc(size_t size){
    intprt_t ptr;
    #pragma offload target(mic:MYDEVICE) out(ptr)
    {
        ptr = (intprt_t) memalign(64, size);
    }
    return ptr;
}

void offload_Free(void* p){
    intprt_t ptr = (intprt_t)p;
    #pragma offload target(mic:MYDEVICE) in(ptr)
    {
        free((void*)ptr);
    }
}
```

<pre>#ifdef USE_MIC dY = (double*) offload_Alloc(ysize*elemSize); #else cublasAlloc(ysize, elemSize, (void **) &dY); #endif</pre>	<pre>if (dAtmp != 0) { #ifdef USE_MIC offload_Free(dAtmp); #else CUBLAS_FREE(dAtmp); #endif dAtmp = 0; };</pre>
--	---

图 26-5 Intel MIC 上的内存分配和释放代码段

卸载模式的另一个不同之处在于其只允许传输一维或者连续内存数据，而 CUDA 的 cudaMemcpy 2D 允许传输二维子矩阵。这种操作可以通过如下循环来替代。

- (i) 打包二维数据到一个连续的一维缓存中。
- (ii) 利用卸载制导语句传输一维连续的缓存。
- (iii) 在设备端恢复原始二维数据。

调用 CUBLAS 中的函数可以简单地替换为 MKL 中对应函数的调用。图 26-6 展示了将 CUBLAS 中 DGEMM 调用等价地转换为 MKL 中 DGEMM 调用的示例。卸载制导语句将指针转换为整数类型传输并在使用前将其重新转化为指针类型。

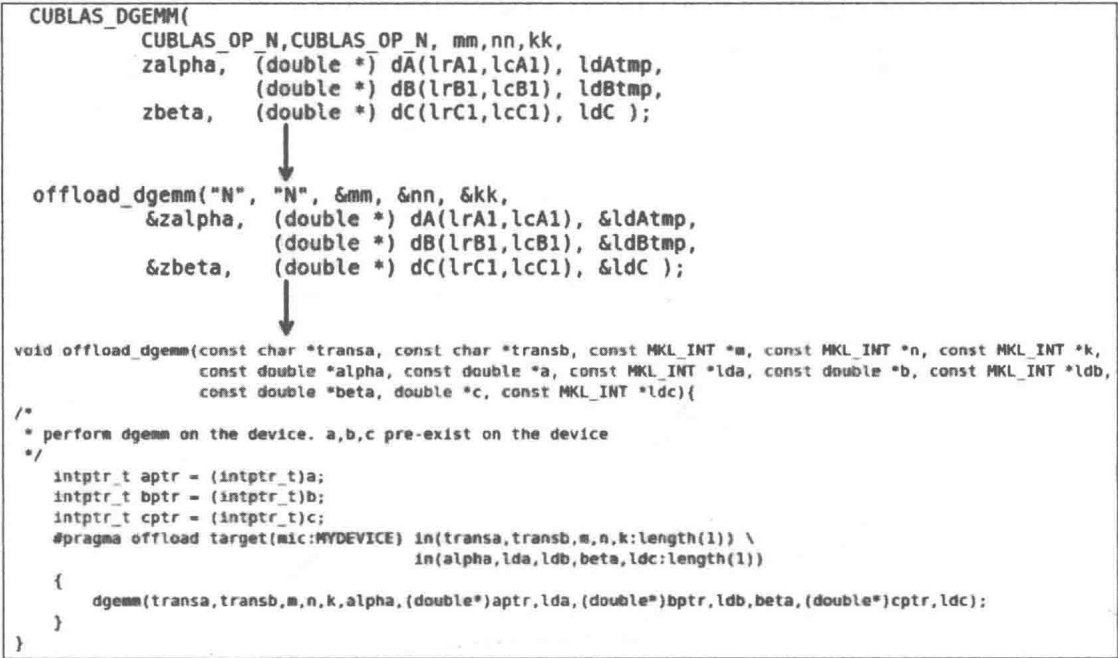


图 26-6 CUBLAS 到 Intel MKL 的转化

26.4 数值结果

核外 Cholesky 分解器的数值实验在 Beacon 集群上执行，其中包括协处理器以验证可扩展性和性能特征。

Beacon 位于美国国家计算科学中心，包括 Intel Xeon 处理器和 Intel Xeon Phi 协处理器，在 2012 年 11 月的 Green500 排行榜中名列第一，能效为 71.4% (2.449GFLOPS/W)。Beacon 包括 48 个计算节点，每个节点中有两个 8 核 Intel Xeon E5-2670 处理器（总共 16 个物理内核），以及 4 个 Intel Xeon Phi 5110P 协处理器。每个协处理器包括 8GB 设备内存以及 60 个频率为 1.053GHz 的内核。CPU 端每个节点上配置 256GB 内存。Beacon 使用 FDR InfiniBand 互联。节点中的每个内核都能访问所有协处理器。GPU 的测试结果在 Keeneland 机器 (<http://keeneland.gatech.edu/>) 上进行。Keeneland 集群包括 264 个节点，总的双精度浮点性能为 615TFLOPS。每个节点配置 32GB 主机端内存、两个 Intel Xeon E5 处理器以及 3 个 NVIDIA M2090 GPU。每个 M2090 GPU 的峰值为 665GFLOPS，并且配置 6GB 设备内存。

表 26-1 总结了 Beacon 上每个节点利用 4 个 MPI 任务执行 LU 分解的性能，每个 MPI 任务都辅助以一个协处理器（设备编号为 MOD(MPI rank, 4)）。OOC 列块大小大约为 6GB 设备内存，块大小为 MB=NB=512。表 26-2 展示了 Cholesky 分解的性能结果。每个协处理器最高性能能达到 370GFLOPS。

表 26-1 Beacon 上 LU 分解的性能

处理器网格	N	每 MIC 的大小 (GB)	每个 MIC 的性能 (GFLOPS)
8×8	176 000	3.9	140
8×8	250 000	7.8	172
8×8	350 000	15.3	208
10×10	250 000	5	124

(续)

处理器网格	N	每 MIC 的大小 (GB)	每个 MIC 的性能 (GFLOPS)
10 × 10	350 000	9.8	171
10 × 10	400 000	12.8	177
12 × 12	250 000	3.5	112
12 × 12	500 000	13.9	141

表 26-2 Beacon 上 LLT 分解的性能

处理器网格	N	列块数	每个 MIC 的性能 (GFLOPS)
1 × 1	156 672	20	372
4 × 1	313 344	20	349
8 × 2	626 688	20	340
2 × 1	156 672	10	311
10 × 10	313 344	10	319
10 × 10	626 688	10	260
4 × 1	163 840	6	281
8 × 2	327 680	6	269
16 × 4	655 360	6	212
16 × 4	983 040	12	246

图 26-7 展示了 Beacon 上每个协处理器的性能与主机和设备内存大小比率的关系。实验在 4 个节点上执行，每个节点上有两个协处理器和两个 CPU 内核参与计算。选择 4*2 处理器网格。块大小 NB 为 512。memsize 固定为 737 280，这等价于 5.625GB 设备内存（每个 MPI 任务配置一个协处理器）。最大的矩阵大小为 $N = 360\,448$ ，总共需要主机端 948GB 内存，每个节点 242GB 内存。最大的 ρ 为 $(360\,448 \times 360\,448) / (737\,280 \times 1024) = 20$ 。曲线的趋势表明， ρ 值越大性能越好。

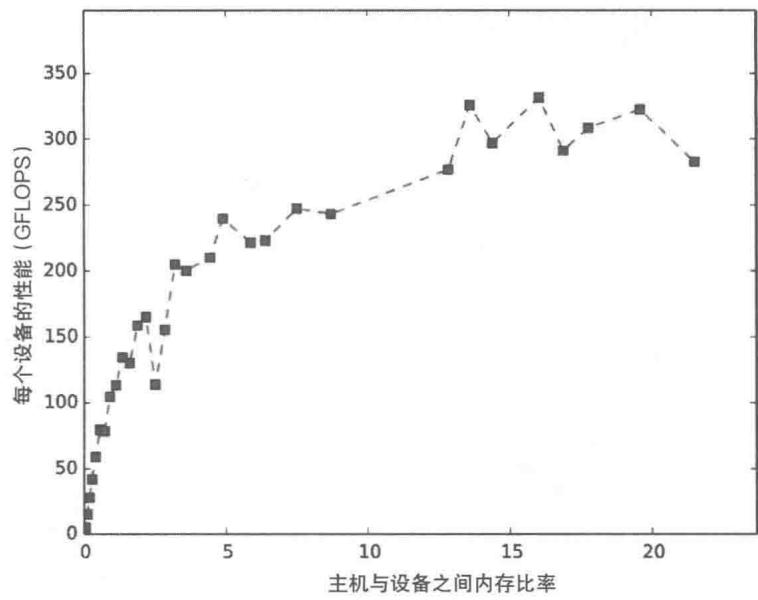


图 26-7 Beacon 上性能与主机和设备之间内存比率 ρ 的关系。处理器网格为 4*2， ρ 最大为 20

图 26-8 和 26-9 展示了不同矩阵大小的性能，因此可以比较其与直接调用 ScaLAPACK 中 PDPOTRF 例程的性能。在 Keeneland 上，矩阵大小最大为 86 400。在一个 GPU 和 8 核 CPU 上比较性能。总体而言，GPU 的性能要优于 CPU。特别是在矩阵规模 N 较大时 GPU 性能持续增加，CPU 端在中等大小的 N 时就已达到最优性能，并且随着 N 的继续增加，性能还显著下降。在 Beacon 上，协处理器的性能也能在较大 N 时维持增长。

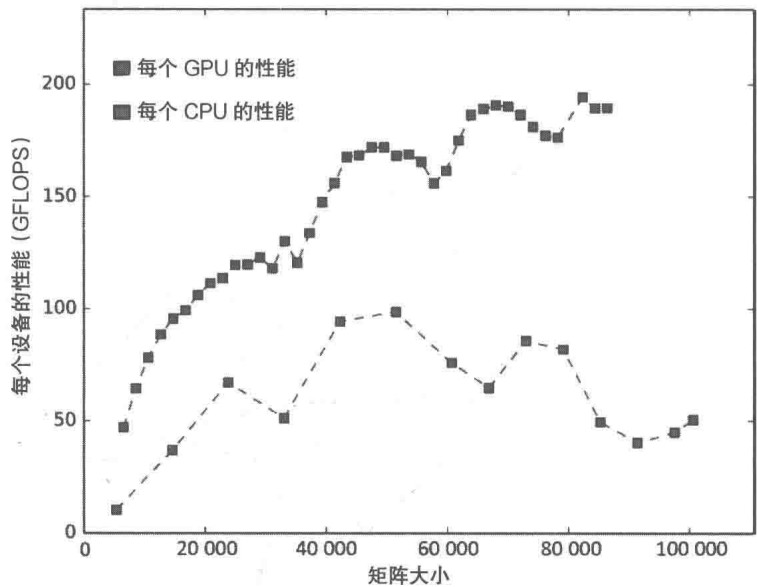


图 26-8 Keeneland 上不同矩阵大小 N 的性能，最大的矩阵大小为 $N = 86\,400$ ，这等价于每个节点使用 28GB 主机内存。图上也展示了仅用 CPU 调用 ScaLAPACK 的性能

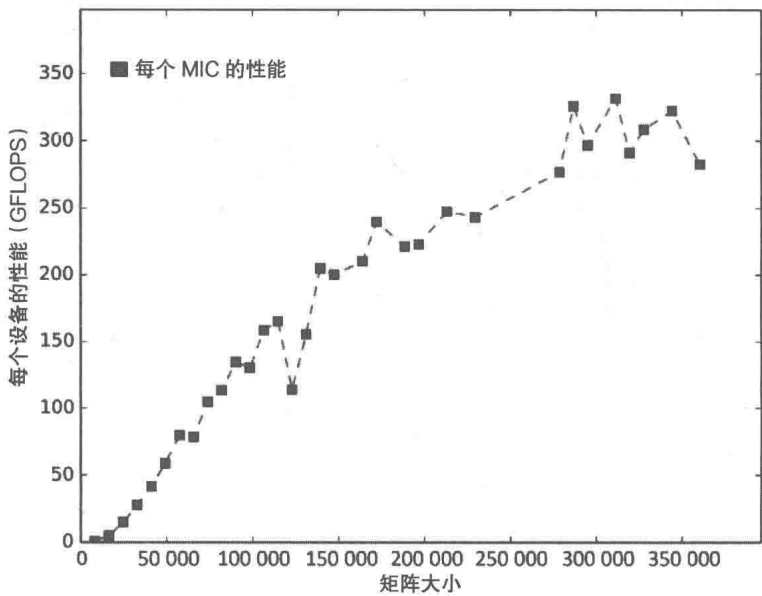


图 26-9 Beacon 上不同矩阵大小 N 的性能，最大的矩阵大小为 $N = 360\,448$ 。每个节点使用 242GB 主机内存

图 26-10 展示了 Keeneland 上固定处理器网格大小的性能数据，研究了两个平台的可扩展

展性，包括弱可扩展和强可扩展。处理器网格是 $4p \times 4p$ ，使用 $2 \times p^2$ 个节点。在弱可扩展性方面，矩阵大小为 $N = 84\,352 \times p$ ，每个 MPI 任务使用 665MB 设备内存，分块大小 $NB=128$ 。总体性能基本上是线性增长，每个 GPU 的性能大约为 170GFLOPS，整体上有轻微的降低。对于强可扩展性，矩阵大小固定为 $N = 84\,352$ ，每个 MPI 任务使用 665MB 设备内存，分块大小 $NB=128$ 。总体性能随着处理器网格大小的增加而增长。总体范围上看执行时间下降。

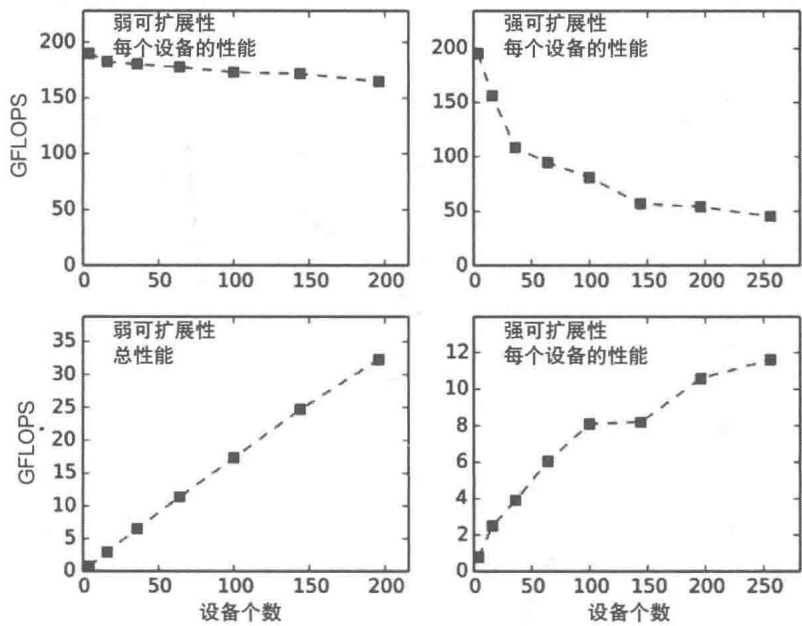


图 26-10 Keeneland 上的弱和强可扩展性。处理器网格是 $4p \times 4p$ ，使用 $2 \times p^2$ 个节点， p 最大为 7

图 26-11 展示了 Beacon 上的强可扩展性。这里也有类似的结果，总体性能随着处理器网格 $p \times p$ 的增加而增长。分解的总体时间持续下降。从这些可扩展性研究结果看，OOC 算法可以获得较好的可扩展性，并且可以扩展到较大的处理器网格上。

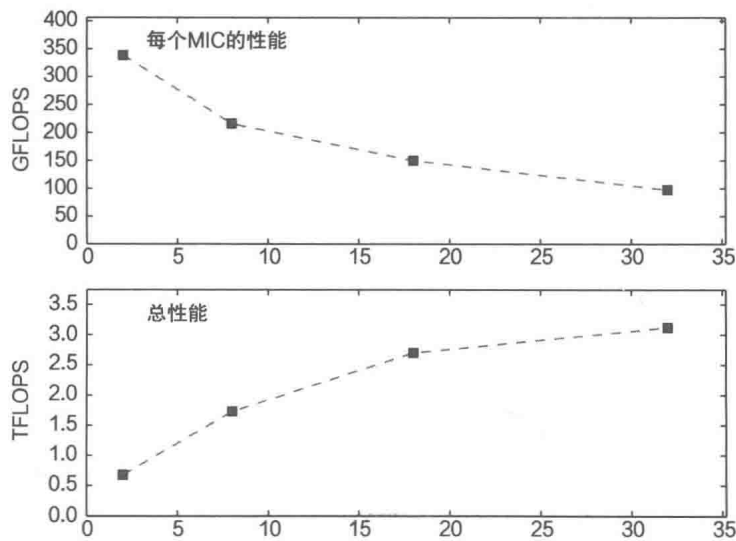


图 26-11 Beacon 上的强可扩展性。处理器网格为 $2p \times p$ ，使用 p^2 个节点， p 最大为 6

26.5 结论和展望

总之,本章介绍了分布式内存上与 ScaLAPACK 接口兼容的并行 LU 和 Cholesky 解法器,并且利用了 GPU 和 MIC 加速器。OOC 算法支持几倍于可用设备内存的较大问题规模的求解,性能随着数据大小设备内存比率 ρ 的增加而增长。

未来的优化将包括利用异步 BLAS 操作以及并发数据传输优化性能。

26.6 致谢

本工作得到了美国国家科学基金、美国能源部、美国橡树岭国家实验室、中国香港中文大学以及美国田纳西大学的支持。本章的内容基于美国国家自然科学基金 0711134、0933959、1041709 和 1041710 的支持以及田纳西大学使用位于美国国家计算科学中心 (<http://www.nics.tennessee.edu>) 的 Beacon 计算资源完成的工作。本项工作利用的 Keeneland 计算设备位于佐治亚理工学院,并得到美国国家自然科学基金 OCI-0910735 项目的支持。作者还得到美国政府 DE-AC05-00OR22725 项目的支持,因此美国政府拥有免费发布和转载本章工作成果的非专有权利,或者允许其他人为了美国政府的目标这样做。

26.7 更多信息

- Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., Tomov, S., 2011. QR factorization on a multicore node enhanced with multiple GPU accelerators. IPDPS 2011, Alaska, USA.
- Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S., 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. J. Phys. Conf. Ser. 180, 012037.
- Barrett, R.F., Chan, T.H.F., D'Azevedo, E.F., Jaeger, E.F., Wong, K., Wong, R.Y., 2010. Complex version of high performance computing LINPACK benchmark (HPL). Concurr. Comput.: Pract. Exper 22 (5), 537-587.
- D'Azevedo, E., Dongarra, J., 2000. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. Concurr. Comput.: Pract. Exper. 12, 1481-1493.
- D'Azevedo, E., Hill, J.C., 2012. Parallel LU factorization on GPU cluster. Proced. Comput. Sci. 9, 67-75.
- Fogue, M., Igual, F.D., Quintana-Orti, E.S., van de Geijn, R., 2010. Retargeting PLAPACK to clusters with hardware accelerators. FLAME Working Note 42.
- Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J., 2010. CULA: Hybrid GPU accelerated linear algebra routines. SPIE Defense and Security Symposium (DSS), April 2010.
- Jetley, P., Wesolowski, L., Gioachin, F., Kale, L.V., Quinn, T.R., 2010. Scaling hierarchical N-body simulations on GPU clusters. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC10. pp. 1-11.

- Quintana-Orti, G., Igual, F.D., Quintana-Orti, E.S., van de Geijn, R.A., 2009. Solving dense linear systems on platforms with multiple hardware accelerators. Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming PPOPP '09, 121-130.
- Song, F., Dongarra, J., 2012. A scalable framework for heterogeneous GPU-based clusters. Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures. ACM, 91-100.
- Song, F., Tomov, S., Dongarra, J., 2012. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. Proceedings of the 26th ACM international conference on Supercomputing. ACM, 365-376.

稀疏矩阵向量乘：并行化和向量化

Albert-Jan N. Yzelman, Dirk Roose, Karl Meerbergen

比利时，鲁汶大学

27.1 引言

稀疏计算在计算代码中十分常见，其中稀疏矩阵向量（SpMV）乘法作为重要的计算核心广泛应用在各个领域中，包括模拟（例如计算流体动力学、结构分析），优化问题（例如经济、运输调度），以及数据分析（例如药物检验、社会网络）等。稀疏矩阵的特点是矩阵中有大部分元素等于零。为了利用稀疏性，这些矩阵存储在专门设计的数据结构中，从而可以避免与零相乘的无用计算。

向量单元的宽度不断增加以及每个内核有效带宽的降低是当今硬件的发展趋势。对于稀疏计算，这两种趋势是相互冲突的。本章将考虑拥有向量处理能力的多核架构上的稀疏矩阵运算，并针对向量化稀疏计算设计一个可用和高效的数据结构。

一个 $m \times n$ 矩阵 A 具有 m 行和 n 列，并包含元素 a_{ij} ，其中 $i = 0, 1, \dots, m-1, j = 0, 1, \dots, N-1$ 。考虑到矩阵向量乘法 $y = Ax$ ，其中 x 向量以及 y 向量的维数分别为 n 以及 m 。输出向量 y 的每个元素由 A 矩阵的一行元素与输入向量 x 的点积得到，即 $y_i = \sum_{j=0}^{n-1} a_{ij}x_j$ ，如图 27-1 所示。若 A 矩阵的所有元素按行优先方式存储，则 A 矩阵一行中的连续元素存储在连续的内存空间中，那么 A 矩阵和 y 向量都连续读取且访问步幅为 1。这样的流访问是非常高效的，从而达到很高的数据移动带宽。非必需的缓存缺失则只出现在输入向量 x 上，因为它的元素需要反复读取。

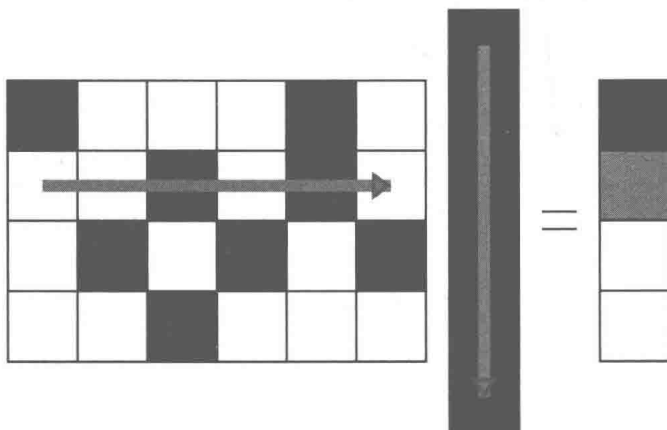


图 27-1 稀疏矩阵向量乘 $y = Ax$ ，图中为 y_2 的计算过程。深灰色的方块代表非零元，而空的方块代表零元

考虑到缓存的大小， A 矩阵一种高效的存储方式是将其划分成一系列子块，且每个子块可以被精确地放到缓存中。这类缓存感知方法的另外一个发展方向是开发存储方案以及算

法，无须考虑缓存层次结构细节即可获得好的性能。这样的缓存无关方法不需要设置依赖于体系结构的参数，如上述子块大小。

如果 A 是稀疏的，只有 $a_{ij} \neq 0$ 的元素和它们在矩阵中的位置应该存储。这些稀疏存储方案将在 27.2 节中介绍。对于压缩的稀疏矩阵结构，每个数据字上的操作次数（也称为计算密度）是非常低的。由于执行浮点运算的时间比读取或存储数据字的时间要短得多，SpMV 乘法性能只能达到峰值性能很低的百分比。这是稀疏计算的特性问题，并且可能导致以下两条结果：

- (a) 稀疏矩阵的压缩存储提高了性能；
- (b) 稀疏计算不能受益于向量化。

我们表明，即使压缩存储会导致额外操作，但由于该操作是带宽受限的，它仍然可以获得好的性能。然而，后者并不总是正确的：尽管 SpMV 乘法具有低计算密度，但向量化仍可以提高其在 Intel Xeon Phi 协处理器上的性能。

27.2 节介绍几种稀疏矩阵数据结构。27.3 节将讨论在共享内存系统上 SpMV 乘法的几种并行化策略。27.4 节介绍如何通过向量化提升其在 Intel Xeon Phi 协处理器上的性能。性能结果将在 27.5 节展示。

27.2 稀疏矩阵数据结构

一个用于稀疏矩阵计算的基本数据结构是坐标 (COO) 格式，该格式将稀疏矩阵 A 存储为三个长度为 nz 的数组 (i, j, k) 。这里， nz 为 A 矩阵非零元的个数。在 COO 格式中，第 k 个非零元对应的三个数组值分别为 $v_k = a_{ij}$ ， $i_k = i$ ， $j_k = j$ 。需要注意的是， A 的非零元可以按任意顺序存储。算法 27-1 使用 COO 计算 $y = Ax$ 。

算法 27-1：基于 COO 的 SpMV 乘法

```
1: for  $k=0$  to  $nz-1$  do
2:   add  $v_k \cdot x_{j_k}$  to  $y_{i_k}$ 
3: end for
```

对于基于 COO 的 SpMV 乘法，计算密度通常在 0.25 和 1 之间，这取决于缓存效率。然而，为了充分利用当前处理器的计算能力，更高的计算密度是必要的。

例如，Intel Haswell E3-1225 处理器具有 4 个 3.2 GHz 内核。该处理器支持 AVX 2 向量指令，这允许 4 个 64 位数据字上的同时（向量）操作。该处理器还支持融合乘加 (FMA) 指令，即在每个数据字上同时执行两个浮点运算 (flop)。在计算处理器峰值性能时，假定使用 FMA 操作，与内核数、可以同时操作的数据字的个数（向量长度）和处理器速度相乘，即 $2 \times 4 \times 4 \times 3.2 = 102.4$ GFLOPS。一个 DDR3-1600 内存控制器的带宽为 12.8 GB/s，或 1.6 千兆字每秒。为了充分利用该处理器的计算能力，对于每个提供给处理器的数据字需要发生 $1024/16=64$ 次计算。

Intel Xeon Phi 7120A 协处理器具有 61 个主频为 1.238 GHz 的内核，同时支持 FMA 指令，并包含可以同时处理 8 个 64 位数据字的向量单元。因此，它的峰值性能为 $2 \times 61 \times 8 \times 1.238 = 1208$ GFLOPS。该协处理器包含 16GB 的 GDDR5 内存，最高带宽为 352GB/s，即 44 千兆字每秒。对于在该处理器上计算受限的算法，在每个数据字上至少需要 28 次操作 ($1208/44=27.5$)。

显然，对于以上讨论的两个现代处理器架构，每个数据字需要的操作数量要远远高于如 SpMV 乘法等稀疏计算算法所能提供的。因此，这里的问题是带宽受限的，即内核需要花费

相对较长的时间等待所需要的数据。

通过更高效的缓存行为，可以在一定程度上解决这个问题。而稀疏矩阵数据结构的访问本身是连续的，并且步幅为 1，向量 \mathbf{x} 和 \mathbf{y} 的访问模式由非零元的顺序确定：一个行优先顺序会带来 \mathbf{y} 向量上步幅为 1 的连续访问，但 \mathbf{x} 向量上的访问是随机的。然而，一个完全随机的顺序将使 \mathbf{x} 和 \mathbf{y} 向量上的随机访问成本十分昂贵。随机访问（相对于连续的访问）具有高延迟成本以及较低的吞吐量，并可能拖延内核的速度。分形存储方案可以优化非零元的顺序，尽可能使 \mathbf{x} 和 \mathbf{y} 向量的内存访问开销最低。空间填充曲线对非零元坐标 (i, j) 进行重新排序。图 27-2 所示为基于希尔伯特曲线的非零元的缓存无关序。

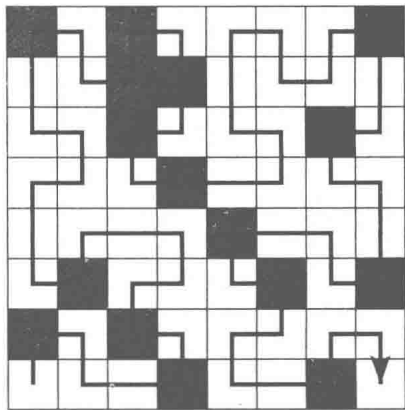


图 27-2 根据希尔伯特空间填充曲线遍历稀疏矩阵

矩阵感知方法可以分析 A 矩阵的非零元结构，并根据该信息来优化数据存储。一些库软件可以自动检测小的结构块，并结合缓存感知的数据结构进行自动调优，这些库软件包括 OSKI 和 pOSKI 等。有关缓存感知以及矩阵感知方法的更多信息可以参阅 Vuduc et al. (2005)。或者，通过稀疏矩阵分割，矩阵的行和列可以重新排序，在一种改进的缓存无关的方式下使得缓存缺失数量的上限最低。自动调优或矩阵感知方法的预处理成本限制了这类方法的可用性。

27.2.1 压缩后的数据结构

COO 格式的数据结构可以通过几种方式压缩。如果首先在非零元上使用行优先顺序，随后的 i 所对应的项具有相同的行号。这些冗余数据可压缩成一个大小为 $m+1$ 的数组 s ，其中 s_i 的值设定为 $0 \leq k < \text{nz}$ 中的最小整数，其中 $i_k = i$ ， $0 \leq i < m$ ， s_m 的值定义为 nz 。在数据结构 (s, j, v) 中，行压缩存储 (CRS) 格式的后两个数组与 COO 相同，其中非零元按行优先的顺序存储。事实上，CRS 是存储非结构化稀疏矩阵的标准^①。需要注意的是，在 CRS 格式下不能实现缓存无关优化，例如图 27-2 所示的分形非零元排序。

一个基于 CRS 格式的 SpMV 乘法包含两层循环，第一层遍历所有的行，第二层遍历每行的非零元，如算法 27-2 所示。

算法 27-2：基于 CRS 的 SpMV 乘法

```

1: for  $i = 0$  to  $m - 1$  do
2:   for  $k = s_i$  to  $s_{i+1} - 1$  do
3:     add  $v_k \cdot x_{j_k}$  to  $y_i$ 
4:   end for
5: end for

```

类似地，使用列优先顺序存储并且压缩 j 会得到列压缩存储 (CCS) 格式。需要注意的是，CRS 需要 $\Theta(2\text{nz} + m + 1)$ 的存储空间，相对于 COO 格式的 $\Theta(3\text{nz})$ ，存储开销显著降低。

压缩存储也可以不受限于一种特定的非零元顺序，如行优先或者列优先。考虑 COO 数据结构，在任意非零元顺序下，相邻数值的 i 与 j 之间的差值可以定义为增量数组，分别表

① CRS 也称为压缩稀疏行 (CSR)。

示为 Δi 以及 Δj ，即

$$(\Delta i)_k = \begin{cases} i_0 & k = 0 \\ i_k - i_{k-1} & k > 0 \end{cases}$$

对于 Δj 也类似。图 27-3（左图）说明了这种增量编码方式。

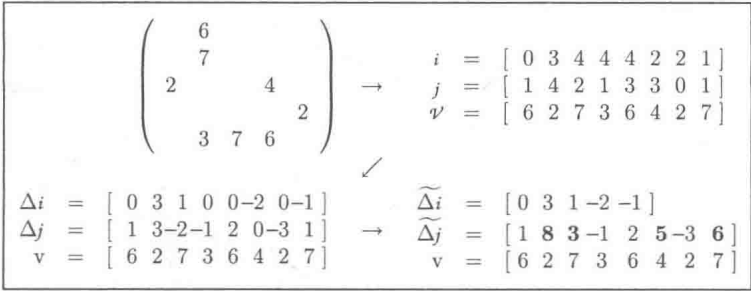


图 27-3 一个例子矩阵的框架图（左上图）和它的 COO 表示（右上图）。通过从 COO 增量编码（左下图）的 Δi 中删除零元，可以推导出 BICRS 数据结构（右下图）。 $\widetilde{\Delta j}$ 中的加黑元素表示在对应的非零元上发生了行跳转；需要注意的是，这表示把 n 加到相应的列增量中

当连续的非零元属于同一行或列时，零值串出现在 Δi 或 Δj 中，这使得压缩存储成为可能。我们将存储 $\widetilde{\Delta i}$ 而不是 Δi ，其中 $(\widetilde{\Delta i})_0 = (\Delta i)_0$ ，同时删除了所有剩余的零元。为了不丢失非零元的行坐标信息，修改了 Δj 数据，即将 n 加到对应元素的列增量数组中，从而在 Δi 中的每次矩阵行跳转时进行标记。由此产生的数组 $\widetilde{\Delta j}$ 可由以下公式决定：

$$\widetilde{\Delta j}_k = \begin{cases} \Delta j_0 & k = 0, \\ \Delta j_k & k > 0 \text{ 且 } \Delta i_k = 0 \\ \Delta j_k + n & k > 0 \text{ 且 } \Delta i_k \neq 0 \end{cases}$$

图 27-3（右图）说明了这种做法。需要注意的是，列增量数组中附加的簿记实质上只在每项上产生一个位开销。

类似于 CRS 和 CCS，行和列的角色可以通过对 Δj 进行压缩而互换。这些数据结构分别称为双向增量 CRS (BICRS) 和 BICCS 格式。BICRS 允许高效的实现，如算法 27-3 所示。BICRS 的引入使得利用空间填充曲线进行数据移动成为可能。更详细的信息请参考 Yzelman and Bisseling (2012)。

算法 27-3：基于 BICRS 的 SpMV 乘法

```
1:  $c = 0, i = \widetilde{\Delta i}_c, j = \widetilde{\Delta j}_0$ 
2: for  $k = 0$  to  $n_z - 1$  do
3:   add  $v_k \cdot x_j$  to  $y_i$ 
4:   add  $\widetilde{\Delta j}_{k+1}$  to  $j$ 
5:   if  $j$  indicates a row jump then
6:     add  $\widetilde{\Delta i}_c$  to  $i$ 
7:     increment  $c$ 
8:   end if
9: end for
```

需要注意的是，分别只需要 $\Theta(\log_2 m)$ 以及 $\Theta(\log_2 n)$ 位来存储 $\widetilde{\Delta i}$ 与 $\widetilde{\Delta j}$ 的一个元素。通过使用短整型来存储这些数组，实现了进一步的压缩存储。我们将该数据结构称作压缩的 BICRS 格式。

27.2.2 分块

使用压缩的 BICRS 格式与稀疏输入矩阵的分块是相互促进的。我们采用了一种两层次混合方法，其中 A 矩阵分为小的方块，并且这些块根据希尔伯特空间填充曲线进行排序。对这些块以上述方式进行排序可以提高缓存效率，并且可以通过 BICRS 格式有效地存储。构建这种更高级别的数据结构是廉价的，因为基于希尔伯特的 COO 格式需要对每个子块而非每个非零元进行计算，这也适用于基于希尔伯特坐标的子块（而非非零元）的排序。

在每个块中，非零元根据行优先顺序存储。我们优化了每个块的维度，使得 $\tilde{\Delta}i$ 与 $\tilde{\Delta}j$ 可以用 16 位（即使用 short int）来存储。块中非零元的压缩存储可通过压缩的 BICRS 格式实现，因为稀疏块中的大多数行都是空的，所以 BICRS 优于 CRS——无论这些行是否为空，后者都需要 $m+1$ 的存储开销。通过借助短整型进一步压缩，这种两层次方案可以确保分块的矩阵存储比原始未对 A 矩阵分块的 CSR 格式存储开销低。更多详细信息请参考 Yzelman and Roose (2014)。

27.3 并行 SpMV 乘法

为了在共享存储体系结构上对上述算法进行并行化，工作负载需要分布在可用的多线程或多进程上。在一般情况下，数据驻留在全局存储器中供全部线程访问。如果在连续的大块内存空间中分配矩阵和向量，则每个线程可以根据给定的工作负载分配访问块中的元素。这样，基于 CRS 的 SpMV 乘法可以很容易地通过 OpenMP 的编译制导语句进行并行化，即在算法 27-2 中的最外层 for 循环（语句 1）前加入编译制导语句 `omp parallel for schedule(dynamic, 8)`。

然而，通过使每一个内核分配自己的内存块，数据也可以显式分配，从而使得每个线程分配在并行计算中它所操作的数据元素。在共享存储器体系结构上使用显式数据分配有以下两方面原因：（1）避免数据竞争；（2）利用数据局部性。显式地分配数据或保持所有数据全局可用可显著影响性能。以下描述了两种 SpMV 乘法的数据分配类型，它们都在不同程度上使用了显式数据分配。

27.3.1 部分分布式并行 SpMV

在部分分布式方式中，如图 27-4 所示，对矩阵 A 按行划分，得到子矩阵 A_s ，其中 $s = 0, \dots, p-1$ ， p 为线程数。每个子矩阵 A 是一个少于 m 行但具有完整 n 列的长方形矩阵。遵守显式分配的思想，每个子矩阵存储在独立的内存块中。为了确保负载均衡，每个 A_s 应含有大约 nz/p 个元素（虽然行数可不同于 m/p ）。每个 A_s 分为有固定的行和列维度的子块，并利用（压缩的）BICRS 格式存储（参照前面章节对分块的描述）。这些子块按希尔伯特填充曲线的定义进行处理。这种缓存无关元素遍历有利于高级别缓存中的数据重用，同时最大限度地减少所需数据的存储开销。

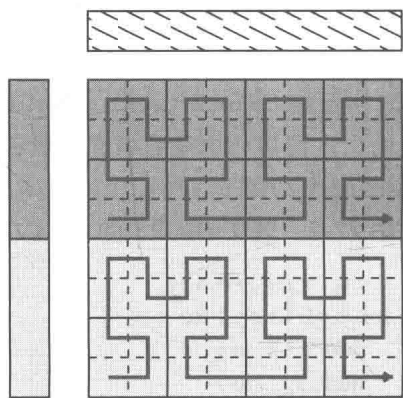


图 27-4 当 $p = 2$ 时的部分分布式方法。把上部和下部矩阵区域分配给不同的线程。该分配保证了每个线程处理大约相同数目的非零元素。因此，在实践中，各区域的高度可能不相同。输出向量（左）是根据矩阵行的分配而分配的。输入向量（顶部）没有显式地分配。上部和下部区域中的希尔伯特曲线表示矩阵块的缓存无关访问模式

根据矩阵行的分配，输出向量 y 也分成 p 个连续且不重叠的块，其中第 s 个块由线程 s 自己分配。线程只访问由它们自己分配的 y_s ，从而避免输出向量上的数据竞争（并发写入），也同时利用 y 向量上的数据局部性。所有线程仍然在整个输入向量 x 上进行操作，该向量尽可能以交错的方式存储在多个存储条中。这在 Intel Xeon Phi 协处理器上是自动完成的，但在其他共享内存体系结构下需要通过“libnuma”库进行人工干预。这可以防止利用 x 向量上的任何数据局部性。根据按行分配的性质及向量 x 的全局可用性，在 SpMV 乘法的执行期间无须显式的线程间通信或同步。更详细的描述请参阅 Yzelman and Roose (2014)。

算法 27-4 是该方法的框架。值得注意的是，最后一行（语句 4）是可选的，并且仅当该应用程序不能使用分布式输出向量时，它才是必需的。然而，要利用数据局部性和提高缓存效率，高效的应用程序通过在每个 y_s 上各自使用线程本地的操作完成 y 向量上的操作。

算法 27-4：部分分布式并行 SpMV 乘法

- 1: Partition A row-wise into local matrices $A_s, s=0, \dots, p-1$ (见图27-4)
- 2: Create corresponding local arrays $y_s, s=0, \dots, p-1$
- 3: Each core $s=0, \dots, p-1$ executes $\text{SpMV}(A_s, x, y_s)$
- 4: Concatenate $y_s, s=0, \dots, p-1$ into y

27.3.2 完全分布式并行 SpMV

在一个完全分布式的方案中，我们不但在矩阵 A 和向量 y 上利用数据局部性，而且在向量 x 上利用数据局部性。在预处理阶段，我们首先通过将稀疏矩阵 A 的非零元划分为 p 个部分，对稀疏矩阵进行分区。这些子块再次形成本地矩阵 A_s ，每个线程可以并行执行本地 SpMV 乘法 $y_s = A_s x_s$ 。然而，在这种情况下， A_s 的行和列可能会重叠， x_s 和 y_s 因此可能会有输入和输出向量的重叠子集。一个好的矩阵划分可以使 A_s 尽可能接近 nz/p 的值，并且使涉及 A_s 、 x_s 和 y_s 重叠部分的通信量最小化。

图 27-5 给出了 4 个线程下双重分隔块对角 (SBD) 格式下的矩阵划分，如文献 Yzelman and Bisseling (2009, 2011) 所述。这种划分可以由 (递归) 超图双向分区得到，这通常需要通过使用矩阵分割器进行预处理。对于这种分割器的例子，可以参考 Vastenhouw and Bisseling (2005) 给出的 Mondriaan 以及 Devine et al. (2006) 给出的 Zoltan。该划分还定义了将 A 矩阵变换为 SBD 格式所需的重排序。重排序产生了 p 个大的本地块 (图 27-5 中的方块)，并用 $p-1$ 个分割十字进行分割。线程在本地块上的局部乘法不需要通信，而涉及分割十字中非零元的乘法运算需要显式的线程间通信。对于包含在分割十字中的矩阵行对应的输出向量元素，为了防止数据竞争，还需要引入同步开销。

线程本地的乘法并没有使用前面章节所

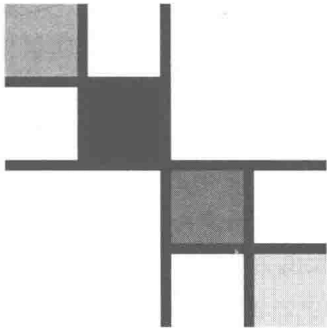


图 27-5 4 个线程下的完全分布式和重排序 SpMV 乘法。每个方块对应于分配给不同线程的子矩阵。分割十字出现在方块之间。在分割十字中的非零元可以划分到任何分割器所跨越的线程。当 x 向量的一个元素与一个分割十字水平重叠时，多个线程可读取该元素；同样，当 y 向量中的一个元素与一个分割十字垂直重叠时，该元素可以被多个线程修改。在实际中，块的大小通常是不相同的

讲的显式分块方案，而是依靠由双重 SBD 重排序自发产生的数据块。这些本地块和分割十字单独存储在压缩 BICRS 数据结构中，其中 $\tilde{A}i$ 与 $\tilde{A}j$ 数组的数据类型是在运行时自动调优的，以达到压缩程度的最大化。

在分布式内存体系结构下，显式地分配所有数据是必需的：现代的超级计算需要完全分布式的算法以实现在多个节点上的并行化，而在节点内则使用其他方法。在此讲述的完全分布式 SpMV 乘法方法在分布式内存以及共享内存体系结构上都是有效的，由 Yzelman et al. (2014) 最近的证明可知。当应用在多插槽共享内存机器上时，它的性能比本节前面所描述的部分分布式并行 SpMV 乘法要高，同时优于其他最新方法。

由于矩阵分割的成本是不可忽略的且随线程数量而增加，因此这种方法不适用于 Intel Xeon Phi 协处理器。然而，当工作负载分布在多个 Intel Xeon Phi 处理器上时，如在典型的分布式内存超级计算环境中，完全分布式是获得良好性能的必然方式。此方法的更多信息请参阅 Yzelman and Roose (2014)。对于本方案的高性能实现，请参考 Yzelman et al. (2014) 基于 C 语言的 Nulticore BSP 实现。

27.4 Intel Xeon Phi 协处理器的向量化

Intel Xeon Phi 协处理器的向量单元作用在向量寄存器上，每个向量寄存器包含 8 个 64 位浮点值。SpMV 乘法的低“浮点运算 - 字节”比率表明这些向量单元不适用于带宽受限的计算；计算是内存受限的，即处理单元会更多地等待所需数据，而不是在这些数据上执行计算。然而，简单地使用上一节描述的部分分布式并行 SpMV 得到的有效带宽远低于可用带宽 352 GB/s，其中线程数量 p 不断增加，直至 $p=240$ 。这表明在协处理器上执行 SpMV 并不是带宽受限的（当然，也不是计算受限的）。相反，在这个架构上的乘法运算已变成延迟受限——使用全部 240 个线程并没有使内存子系统饱和。一种在相同线程数量下可以产生更多数据请求的方法是使用向量化。

使用压缩的 BICRS 格式，实现串行 SpMV 乘法的自动向量化是不可能的。指针运算和间接向量元素寻址阻碍了自动向量化。要使向量化可用，BICRS 乘法算法需要重写为在 $p \cdot q$ 非零元的连续块上进行操作，其中 $p \cdot q = l$ 是向量化的长度，从而使向量寄存器可以同时在这 l 个非零元上进行操作。

我们将 p 和 q 不同的可能选择称为“块大小”，在协处理器上， $p \cdot q = 8$ ，因此出现了 4 种可能的块大小： 1×8 ， 2×4 ， 4×2 以及 8×1 ，如图 27-6 所示。如果 y 、 v 和 x 是对应向量寄存器的长度为 8 的数组，那么 SpMV 乘法的内部计算（即 $y_i = a_{ij}x_j$ ）可以写为一个向量化的 FMA 运算 $y := y + v * x$ 。为了使用这种类型的向量化，重写 BICRS 格式的 SpMV 乘法算法需要使用协处理器上的收集与分发指令，收集指令允许把一个数组中的非连续元素读入一个向量寄存器中，而分发指令则相反。使用这些原语，下面的伪代码说明了向量化的 BICRS 乘法。

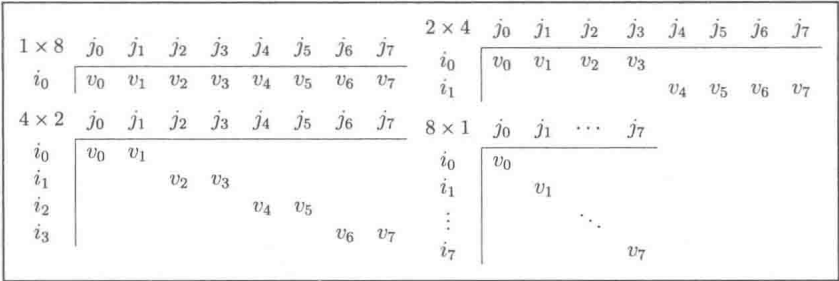


图 27-6 协处理器上 4 个可能的块大小： 1×8 ， 2×4 ， 4×2 以及 8×1


```
1: while there are nonzeros remaining do
2:   get the next set of  $p$  rows to operate on
3:   gather the corresponding output vector elements in  $y$ 
4:   while there are nonzeros remaining on any of the current rows do
5:     get the next set of  $p \cdot q$  (possibly overlapping) column indices
6:     gather the corresponding input vector elements in  $x$ 
7:     retrieve the next set of  $p \cdot q$  nonzero values  $v$ 
8:     do the vectorized multiply-add  $y := y + v \cdot x$ 
9:   end while
10:  scatter the cached output vector elements  $y$  back to main memory.
11: end while
```

在处理 p 行中每行包含的全部非零元时， y 向量的 l 项需要通过求和到 p 个项中归约。例如，对于 2×4 的分块， y 向量的上半部分对应于图 27-6 中的 i_0 ，而下半部分对应于 i_1 。在 1×8 的情况下，则需要一个全归约并将结果写入 i_0 ，而 8×1 的分块则不需要任何归约操作。使用其他的块大小则会导致部分归约。

该算法同样需要对 BICRS 数据结构进行简单修改：对于每一个 $p \times q$ 块，最后 $pq-1$ 个非零元的索引是相对于该块中的第一个元素的。图 27-7 说明了这一原则。当 A 矩阵中没有足够的非零元来填充对应固定行数的块时，显式的零元需要填充，这种填充导致了额外的存储开销和显式地与零元相乘，而这些开销有望被有效的带宽使用所抵消。

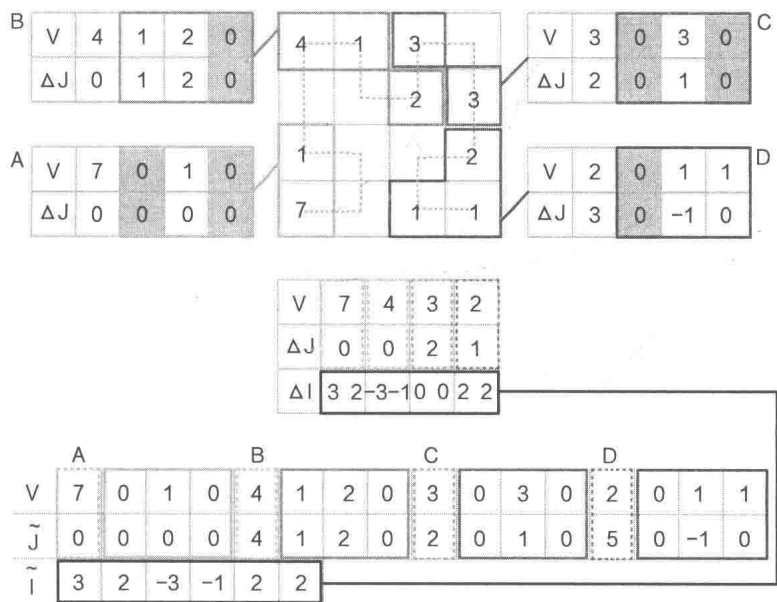


图 27-7 最终向量化的 BICRS 数据结构。它将每块中 $pq-1$ 个非零元的相对编码以及每个块中第一个非零元的 BICRS 编码分离开来，将这两个编码相组合即构成了最终的向量化数据结构。该图说明了 4×4 矩阵上的 2×2 分块，其中非零元按希尔伯特曲线排序。这产生了 4 个数据块，并包含 6 个显式零元（填充）

向量化 SpMV 内核的实现

图 27-8 和图 27-9 提供了向量化的 BICRS SpMV 乘法的一个通用 C++ 实现。在这些代码段中，变量 x 、 y 、 l 、 p 、 q 如文中定义。模板变量 $_i_type$ 和 $_v_type$ 分别对应于索引类型

和数值类型，指向数组 \tilde{d}_i 与 \tilde{d}_j 的指针分别命名为 `row_index_array` 和 `col_index_array`，指向 v 的指针表示为 `value_array`。所有其他变量都在代码段内声明。图中所示的代码是可用的，且可以直接从本书配套发布的代码中获得。^②

```
void spmv( const double *__restrict__ x, double *__restrict__ y ) {
    //declare buffers
    __declspec(align(32)) i_type c_ind_buffer[ 1 ];
    __declspec(align(32)) i_type r_ind_buffer[ 1 ];
    __declspec(align(32)) v_type input_buffer[ 1 ];
    __declspec(align(32)) v_type outputbuffer[ 1 ];
    __declspec(align(32)) v_type outputvector[ 1 ];

    //shift input vector, output vector to first nonzero
    x += *col_index_array; y += *row_index_array;

    //fill buffers and load first l output elements
    for( size_t i = 0; i < l; ++i ) {
        c_ind_buffer[ i ] = *col_index_array++;
        r_ind_buffer[ i ] = *row_index_array++;
        outputbuffer[ i ] = 0;
        outputvector[ i ] = y[ r_ind_buffer[ i ] ];
    }

    //reset start column index, start row index
    c_ind_buffer[ 0 ] = 0; r_ind_buffer[ 0 ] = 0;

    //keep track of how many row blocks in outputvector were processed
    size_t blockrow = 0;
    ...
}
```

图 27-8 通用的体系结构、数据类型和块大小下，向量化 BICRS SpMV 乘法的 C++ 实现。这是内核初始化代码，后续代码清单请见图 27-9

```
...
//start kernel
while( value_array < value_array_end ) {
    //process row
    while( x < x_end ) {
        //fill input buffer (gather)
        for( size_t i = 0; i < l; ++i ) input_buffer[ i ] = x[ c_ind_buffer[ i ] ];

        //do FMA
        for( size_t i = 0; i < l; ++i ) outputbuffer[ i ] += value_array[ i ] * input_buffer[ i ];

        //shift nonzero vector
        value_array += l;

        //shift input vector
        x += *col_index_array;

        //fill c_ind_buffer
        for( size_t i = 0; i < l; ++i ) c_ind_buffer[ i ] = *col_index_array++;
        c_ind_buffer[ 0 ] = 0;
    }
}
```

图 27-9 通用的体系结构、数据类型和块大小下，BICRS SpMV 乘法内部核心代码的 C++ 实现。初始化代码见图 27-8

② 最新版本的代码可参考 <http://albert-jan.yzelman.net/software#SL>。


```

//reduce l outputbuffer elements to p row contributions
for( size_t i = 0; i < p; ++i )
    for( size_t j = 0; j < q; ++j )
        outputvector[ p*blockrow + i ] += outputbuffer[ i*q + j ];

//prepare for next block of p rows
++blockrow;
for( size_t i = 0; i < l; ++i ) outputbuffer[ i ] = 0;

//undo row change signal, shift input vector
x -= n;

//if all elements of outputvector were updated
if( blockrow == q ) {

    //write back outputvector
    for( size_t i = 0; i < l; ++i ) y[ r_ind_buffer[ i ] ] = outputvector[ i ];

    //shift output vector to new row
    y += *row_index_array;

    //load new r_ind_buffer
    for( size_t i = 0; i < l; ++i ) r_ind_buffer[ i ] = *row_index_array++;
    r_ind_buffer[ 0 ] = 0;

    //read new outputvector
    for( size_t i = 0; i < l; ++i ) outputvector[ i ] = y[ r_ind_buffer[ i ] ];

    //reset block row counter
    blockrow = 0;
}
//write back any modified items outputvector may hold
for( size_t i = 0; i < l; ++i ) y[ r_ind_buffer[ i ] ] = outputvector[ i ];
}
}

```

图 27-9 (续)

针对特定体系结构以及块大小的特定代码比基于编译优化的通用实现的性能更好。例如，当 p 非常小时，在输出向量上使用收集和分发原语可能没有意义。图 27-10 显示了在 Intel Xeon Phi 上针对 1×8 数据块的一个特定实现，该代码通过 ICC 内置函数的 AVX-512 指令集进行优化。它假定使用 32 位索引类型 (`_i_type = int32_t`)，64 位浮点值 (`_v_type = double`)，并通过标量 FMA 将结果写回 y 向量。

```

void spmv( const double *__restrict__ x, double *__restrict__ y ) {
    __m512d input_buffer, value_buffer, outputbuffer;
    __m512i c_ind_buffer, zeroF;
    zeroF = _mm512_set_epi32( 1, 1, 1, 1, 1, 1, 1, 0 );
    outputbuffer = _mm512_setzero_pd();

    //load in column indices of the first block, and set c_ind_buffer[0]=0
    c_ind_buffer = _mm512_load_epi32( col_index_array );
    c_ind_buffer = _mm512_mullo_epi32( c_ind_buffer, zeroF );

    //shift vector pointers to the first nonzero position
    x += *col_index_array; col_index_array += l;
}

```

图 27-10 基于 ICC 内置函数的向量化 BICRS SpMV 乘法的 C++ 实现，硬件平台为 Intel Xeon Phi。假设使用 64 位浮点数、32 位索引，以及大小为 1×8 的数据块


```

y += *row_index_array++;

//start kernel
while( value_array < value_array_end ) {
    //process current row
    while( x < x_end ) {
        //gather input vector elements
        input_buffer = _mm512_i32logather_pd( c_ind_buffer, x, 8 );

        //load (stream) nonzero values
        value_buffer = _mm512_load_pd( value_array ); value_array += 1;

        //do FMA; outputbuffer += value_buffer * input_buffer
        outputbuffer = _mm512_fmadd_pd( value_buffer, input_buffer, outputbuffer );

        //load in next block, shift input vector
        c_ind_buffer = _mm512_load_epi32( col_index_array );
        c_ind_buffer = _mm512_mullo_epi32( c_ind_buffer, zeroF );
        x += *col_index_array; col_index_array += 1;
    }
    //write out local contributions (via allreduce), and reset
    *y += _mm512_reduce_add_pd( outputbuffer );
    outputbuffer = _mm512_setzero_pd();

    //shift input vector back to a valid position
    x -= n;

    //shift output vector to next row position
    y += *row_index_array++;
}
}

```

图 27-10 (续)

针对 Xeon Phi 的代码随本书发布，并采用较短的索引类型 (`_i_type = int16_t`) 进一步对稀疏块进行压缩。`c_ind_buffer` 的一次加载可以读取 16 个整型数据，而不是 8 个，这是因为 512 位寄存器可以包含 16 个短整数。这需要图 27-10 中的内部 while 循环展开为两部分，其中第一部分使用 `c_ind_buffer` 的上半部分。之后是一段处理行跳转的代码，随后 `c_ind_buffer` 的上半部分与下半部分（其中包含剩余的未处理的列增量）进行交换，之后则重复这段内部核心代码。手动展开结束后，则继续循环并读入接下来的 16 列增量。当在输出向量上使用收集和分发指令且 $p < l$ 时，这种展开也是必需的：当 $1 < p < l$ 时，部分归约需要输出缓冲区上的额外置换以及相加操作，而将获得的 p 份中间结果累加到输出向量上需要屏蔽的向量加法，这需要通过展开提高效率。为简便起见，这些代码并不在此介绍，要查看展开代码的细节，请参阅所发布的代码。

27.5 评估

我们比较了三个并行 SpMV 算法的执行速度，并用 GFLOPS 衡量，即基于 OpenMP 的 CRS SpMV 乘法，通过 PThreads 实现的部分分布式并行 SpMV 方法（本地矩阵使用压缩 BICRS 格式存储），以及同样用 PThreads 实现的部分分布式方法的向量化版本（使用如前一节给出的向量化的 BICRS 数据结构）。我们同 11.1.1 版本的 Intel Math Kernel Library 进行了对比，其中稀疏矩阵是基于 CRS 格式的。所有的代码使用 14.0.1 版本的 Intel C / C++ 编译器 (ICC) 进行编译。

我们在不同维度以及稀疏度的 6 个矩阵（见图 27-11）上进行了测试。所选矩阵代表 4 类常见矩阵：它们的维度或小（对应的向量可以存放在组合的缓存中）或大，非零元分布式结构化（非零元结构本身有利于缓存重用）或非结构化。依照上述方式对现实世界中更为广泛的矩阵进行分类请参考 Yzelman and Roose（2014）。

小矩阵	行	列	非零元	
nd24k	72 000	72 000	28 715 634	U
s3dkt3m2	90 449	90 449	1 921 955	S
大矩阵				
Freescape1	3 428 755	3 428 755	17 052 626	S
wiki07	3 566 907	3 566 907	45 030 389	U
cage15	5 154 859	5 154 859	99 199 551	S
adaptive	6 815 744	6 815 744	13 624 320	U

图 27-11 在实验中使用的矩阵。在水平分隔线以上的矩阵被视为小矩阵，其相应的输入、输出向量足够小，使其可以被现代体系结构的（组合的）大多数二级缓存所容纳。最右边一列表示，在基于 CRS 格式的 SpMV 乘法中该稀疏矩阵是否拥有一个缓存友好的结构（S 代表结构化的），如果没有这样的结构，则用 U 表示非结构化

27.5.1 Intel Xeon Phi 协处理器

我们在 Intel Xeon Phi 7120A 协处理器上运行实验。该处理器包含 61 个 1.238 GHz 内核和 16GB 内存。每个内核配备了 32KB 一级缓存和 256KB 二级缓存，每个内核支持 4 个硬件线程。除了留给操作系统的内核外，我们开启了所有内核上的全部线程，因此共计 240 个线程。

图 27-12 给出了部分分布式的并行 SpMV 乘法的性能，其中稀疏矩阵格式为向量化的 BICRS。我们使用每个矩阵上所有可能的块大小（1×8，2×4，4×2 或 8×1）进行实验。这些选择导致不同的填充量，如图 27-13 所示。通过对比结果，当选择填充量最小的块大小时，几乎总是可以得到最快的执行速度。唯一的例外是对于 wiki07 矩阵，其中向量化的性能比非向量化的 SpMV 乘法要差。实验结果表明，向量化对 Intel Xeon Phi 协处理器上的性能影响很大，尤其是小矩阵（例如 nd24k 和 s3dkt3m2），其中线程本地的向量部分可以被本地缓存所容纳。

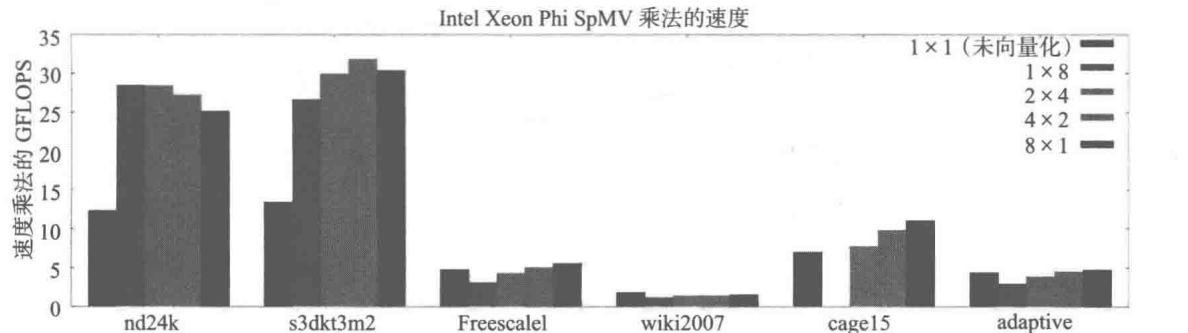


图 27-12 采用向量化 BICRS 的部分分布式方法的性能，其中 4 种可能的分块大小为 1×8，2×4，4×2 以及 8×1。其中，非向量化部分分布式方法的测试结果对应于 1×1 的块大小。cage15 矩阵缺少 1×8 分块的性能值，在这种情况下，额外的填充导致协处理器内存不足，从而使得程序无法成功运行

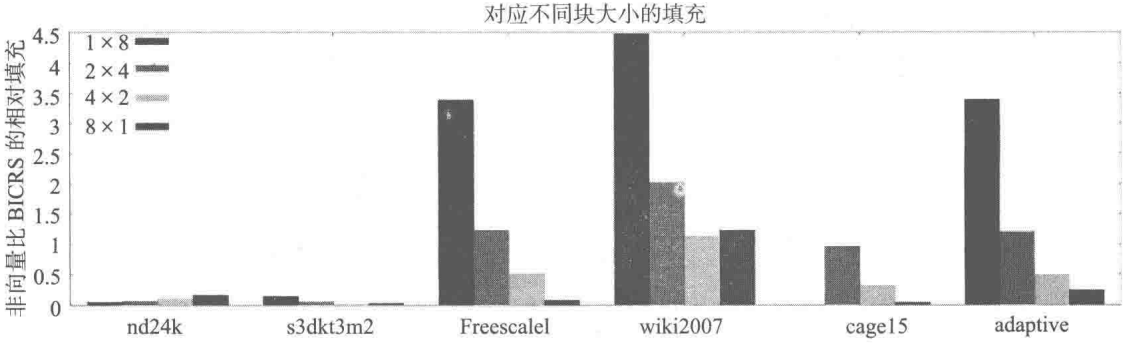


图 27-13 对应于图 27-12 中的矩阵和块大小的相对填充。所展示数据是显式增加的零元数量与原矩阵中所包含非零元数量的比值。cage15 矩阵缺少 1 × 8 分块对应的数值，这是由于增加的填充（大约是 2 倍）导致协处理器内存不足

图 27-17 所展示的性能不包括预处理所需的时间。对于部分分布式方法，预处理的开销相对于构建 CRS 数据结构而言可以忽略不计，因为后者需要对所有的非零元进行排序。在得到实际矩阵结构前，确定对所有可能块大小的填充的先验可以通过对输入矩阵的一次遍历完成。

为了评估上面所讨论的各种优化的性能提升，我们测量了 OpenMP 版本的 CRS 格式实现的性能，依次叠加使用以下优化。

- 1. 部分分布式的 A 矩阵以及 x 向量。
 - 2. 基于希尔伯特曲线排序的缓存无关稀疏矩阵分块。
 - 3. 本地 A 矩阵的向量化 BICRS 存储。
- 这些结果在图 27-14 中展示。

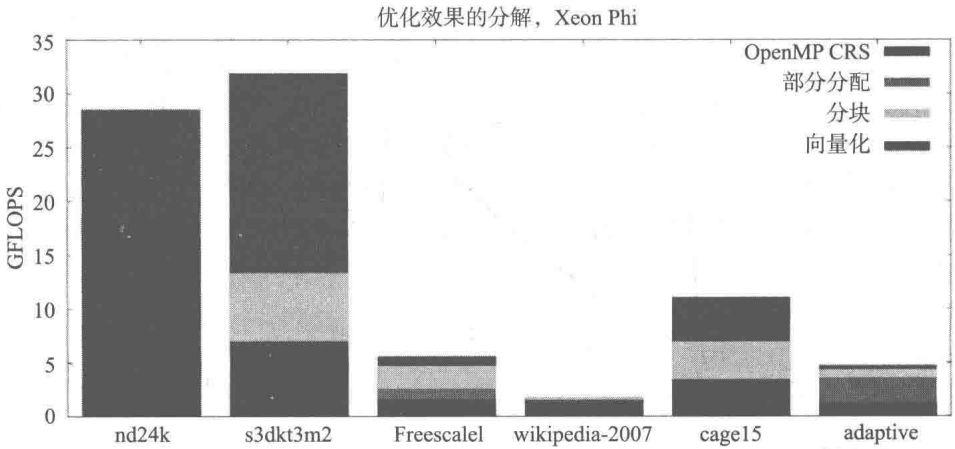


图 27-14 本章讨论的各种优化方法在 Intel Xeon Phi 上带来的性能提升。性能基准是 OpenMP 版本的 CRS 实现，依次叠加使用：（1）部分数据分配，（2）基于希尔伯特曲线排序的稀疏矩阵分块，（3）向量化 BICRS 存储

最有效的优化手段取决于矩阵的维度以及类型。例如，在 nd24k 矩阵上，部分数据分配和分块不能提升 OpenMP 版本的 CRS 基准程序性能，而向量化将其性能提升了约 50%。与此相反，adaptive 矩阵上的主要性能提升来自于部分数据分配，而对于 cage15 矩阵，其主要性能提升来自于基于希尔伯特曲线排序的分块以及向量化两方面因素。

27.5.2 Intel Xeon 处理器

另一个测试平台是一个双插槽 CPU 处理器，它包含两个 Intel Xeon E5-2690 v2 的 “Ivy Bridge EP” 处理器。每个处理器包含 10 个内核，每个内核配备了 32KB 一级缓存以及 256KB 二级缓存，10 个内核共享 25MB 三级缓存。这两个插槽连接到本地内存条中，共计 64GB 1600MHz 的 DDR3 内存。因此最大带宽达到 12.8GB/s 的两倍。

Ivy Bridge 处理器支持 256 位寄存器的向量指令，因此当采用双精度浮点数据时， $l=4$ 。但是 AVX 不支持聚集和分发指令，所以在此架构上的向量化方案很可能比 Intel Xeon Phi 效果要差。收集指令在支持 AVX2 的处理器（Haswell 架构）上可用，而分发指令计划在 AVX3 中支持。

实验测试采用 40 个线程，因此在这种处理器上是开启超线程功能的。图 27-15 给出了双插槽机器下向量化 SpMV 乘法性能。图 27-16 展示了各种优化方法在此架构上的效果。

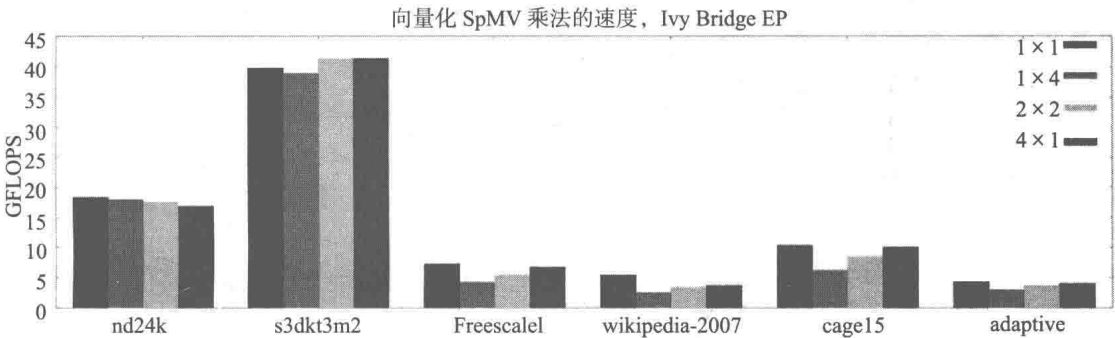


图 27-15 采用向量化 BICRS 的部分分布式方法的性能，测试平台为双插槽 Intel “Ivy Bridge EP” 处理器。在此展示了每个可能的分块大小（1×4，2×2 和 4×1）的性能，并与非向量化的 BICRS（1×1）性能结果进行了对比

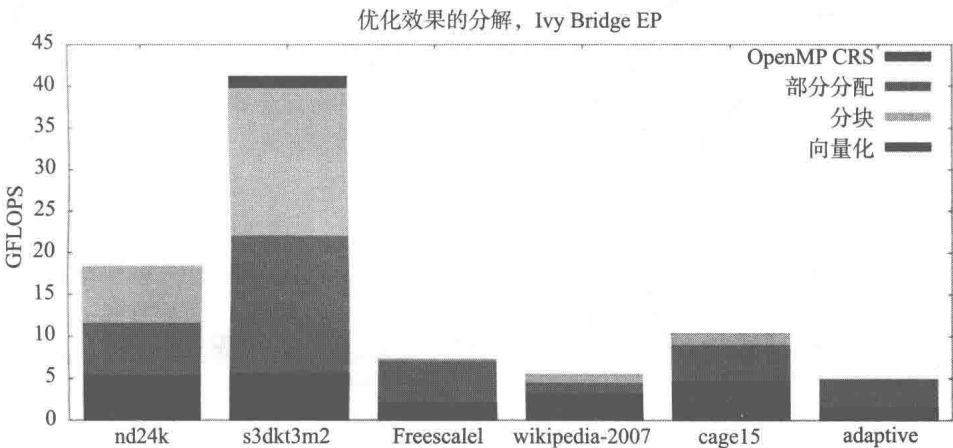


图 27-16 本章讨论的各种优化方法所带来的性能提升，测试平台为双插槽 Intel “Ivy Bridge EP” 处理器。性能基准是 OpenMP 版本的 CSR 实现，依次叠加使用：（1）部分数据分配，（2）基于希尔伯特曲线排序的稀疏矩阵分块，（3）向量化 BICRS 存储

与 Intel Xeon Phi 处理器上的结果类似，每种优化方法的优化效果在很大程度上取决于矩阵的大小以及类型。然而，在一般情况下，向量化的优化效果不如协处理器上的效果显著。这是由于硬件不支持聚集和分发指令，以及 SpMV 乘法在 CPU 上是带宽受限的这一事

实（并非是和协处理器一样的延迟受限）。

27.5.3 性能比较

图 27-17 在 Xeon Phi 上将向量化的 SpMV 性能与以下其他方法进行了对比：（1）直接的 OpenMP 版本的 CRS 解决方案，（2）使用非向量化 BICRS 的部分分布式解决方案，（3）Intel MKL 中的 SpMV 乘法代码。对于测试集中的大矩阵，非向量化的部分分布式算法以及 MKL 的 SpMV 代码性能明显优于简单的基于 OpenMP 并行化的 CRS 代码。指令向量化显著提升了部分分布式算法的性能，且超过了 Intel MKL 中 CRS 格式的 SpMV 乘法的性能，其中一个例外是非结构化的 wiki07 矩阵。

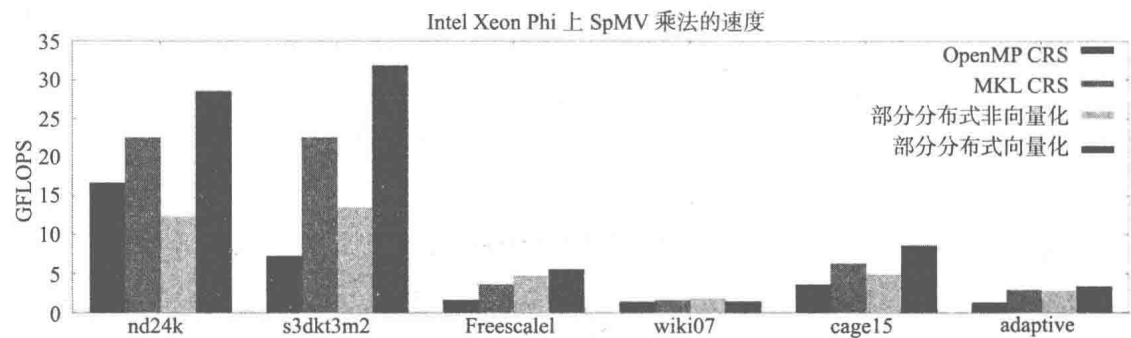


图 27-17 在 Intel Xeon Phi 7120A 协处理器上 4 种并行 SpMV 乘法实现的性能，共使用 60 个内核（240 个线程）：（a）启用 OpenMP 的并行 CRS，（b）启用 MKL 的并行 CRS，（c）使用压缩 BICRS 的部分分布式方法，（d）使用向量化 BICRS 的部分分布式方法

图 27-18 在双插槽“ Ivy Bridge EP”机器上进行了类似的性能对比。向量化在此架构上并不能总是获得较好的性能，而非向量化的部分分布式方法性能优于其他优化方案，其中结构化的 s3dkt3m2 矩阵是个例外（该矩阵通过向量化获得了性能提升）。

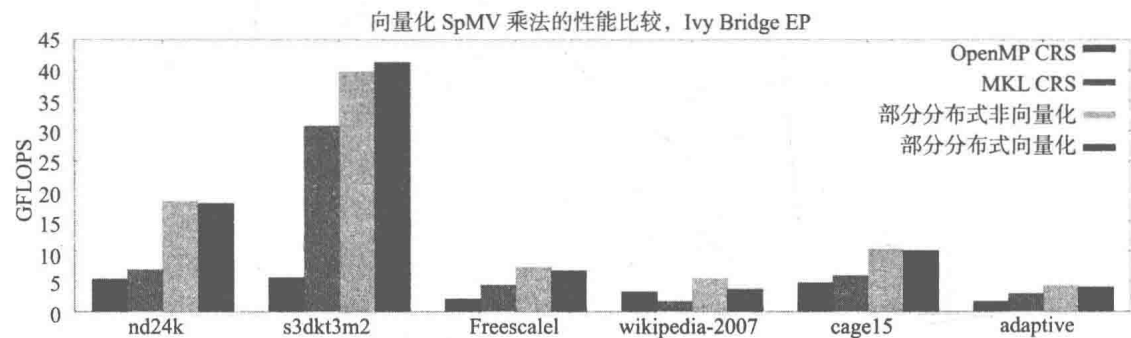


图 27-18 在双插槽 Ivy Bridge 机器上启用 40 个线程测试并行 SpMV 实现的性能：（a）启用 OpenMP 的并行 CRS，（b）启用 MKL 的并行 CRS，（c）部分分布式 BICRS，（d）部分分布式的向量化 BICRS

27.6 总结

由于低“浮点运算 - 字节”比率以及低效率的缓存使用，SpMV 乘法获得好的性能是非常困难的。对于 Intel Xeon Phi 协处理器，一个额外的困难是高延迟的内存访问。

针对共享内存系统上的并行 SpMV 乘法，本章讨论了几种最大化数据局部性以及数据重用的优化策略。集中讨论了支持向量化操作的数据结构、缓存无关的任意非零元遍历，以及基于分块的高级压缩存储技术。还对共享内存以及分布式内存平台上的并行方案进行了讨论。

实验测试在 Intel Xeon Phi 协处理器以及双插槽 Ivy Bridge 两个共享内存系统上进行。结果表明，共享内存并行化技术再加上稀疏矩阵分块、缓存无关遍历、压缩和向量化等优化方法，性能超出了基于 CRS 的 OpenMP 版本的算法实现以及 Intel MKL 库中的 SpMV 代码，对相同编程模型的优化可以获得两个处理器架构上的高效代码。

27.7 致谢

这项工作由 Intel 以及佛兰德斯科技创新促进研究所 (IWT) 资助。

27.8 更多信息

- 稀疏矩阵库：<http://albert-jan.yzelman.net/software#SL>。
- C 语言实现的 MulticoreBSP：<http://www.multicorebsp.com>。
- Devine, K.D., Boman, E.G., Heaphy, R.T., Bisseling, R.H., Catalyurek, U.V., 2006. Parallel hypergraph partitioning for scientific computing. In: Proceedings IEEE International Parallel and Distributed Processing Symposium 2006. IEEE Press, Long Beach, CA.
- Vastenhouw, B., Bisseling, R.H., 2005. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47 (1), 67-95.
- Vuduc, R., Demmel, J.W., Yelick, K.A., 2005. OSKI: a library of automatically tuned sparse matrix kernels. J. Phys. Conf. Series 16, 521-530.
- Yzelman, A.N., Bisseling, R.H., 2009. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. SIAM J. Sci. Comput. 31 (4), 3128-3154.
- Yzelman, A.N., Bisseling, R.H., 2011. Two-dimensional cache-oblivious sparse matrix-vector multiplication. Parallel Comput. 37 (12), 806-819.
- Yzelman, A.N., Bisseling, R.H., 2012. A cache-oblivious sparse matrix-vector multiplication scheme based on the Hilbert curve. In: Günther, M., Bartel, A., Brunk, M., Schöps, S., Striebel, M. (Eds.), Progress in Industrial Mathematics at ECMI 2010, Mathematics in Industry. Springer, Berlin, Germany, pp. 627-633.
- Yzelman, A.N., Roose, D., 2014. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. IEEE Trans. Parallel Dist. Syst. 25 (1), 116-125.
- Yzelman, A.N., Bisseling, R.H., Roose, D., Meerbergen, K., 2014. MulticoreBSP for C: a high-performance library for shared-memory parallel programming. Int. J. Parallel Programm. 42, 619-642.

基于 Morton 排序的性能优化

Kerry Evans

美国, Intel 公司

28.1 通过数据重排提高缓存局部性

为了充分利用处理器的计算能力,现代存储层次尽最大可能高效和及时提供所需数据。但是如果程序中的数据结构忽略缓存时性能会如何呢?这时通常需要进行数据重排和分块(blocking 或称为 tiling),但这样通常会生成一段难以阅读和维护的代码,并且与某种机器绑定而无法实现性能可移植。程序员已熟知多维矩阵中数据的存储或者是行优先格式或者是列优先格式,但是如果程序数据的访问模式非这两种情况时性能会如何呢?例如程序在处理某个元素时需要访问其相邻的 4 个或 6 个元素,而非一整行的 16 个元素;又如按行优先格式访问按列优先格式存储的矩阵;更极端的情况下,需要在不同的方向上遍历数据,而具体顺序依赖于某些历史计算数值。当缓存无法容纳程序所有数据时,性能会变得更为糟糕。

更具算法计算特征地优化多维数据存储方式方面已经有很多工作,数据的访问索引计算十分复杂并且需要特殊硬件或指令来验证其性能效果。

本章研究映射多维数据到一维并保证数据局部性的方法(包括 Morton 排序和 Z 曲线排序),并研究其在矩阵转置和矩阵乘法这两个常用的线性代数问题中的实际效果。我们将优化矩阵转置和矩阵乘法代码使之利用 Intel Xeon 处理器和 Intel Xeon Phi 协处理器,包括考虑排序、向量化硬件以及多线程等特征。本章最后对所有结果进行总结。

28.2 性能改进

性能改进可以从许多方面入手,但是最初的三个主要问题是内存访问、向量化和并行化,我们必须在这三个方面优化到极致才能获得最高性能。

当所需要的数据位于缓存中时,才能最快地传输到 CPU。因为缓存与内存的数据交换是以缓存行为单位的(不是字节或者字),所以如果程序所需要的数据位于不同的缓存行,即使所有的数据都位于缓存中,也会导致性能下降。

例如 Intel Xeon Phi 协处理器上 512 位的向量寄存器,一个缓存行(64 字节)可以填满整个向量寄存器,因此,一次缓存读操作可以加载 16 个单精度浮点数据。这种方式十分高效,但是必须要求所有数据均位于一个缓存行内,否则,需要多次读写才能装满一个向量寄存器。

并行化也是高效利用处理器性能的必选项。如果设计了一种有效的数据分块策略,它能有效提高数据访问性能并利于向量化实现,那么通常该策略也能提供一种划分计算的方式使之高效利用硬件的多线程进行并行处理。

G. M. Morton 提出了 Morton 排序(也成为 Z 排序或 Morton 编码),通过交替每个维度的坐标位,能将多维数据映射到一维并保证数据的局部性。图 28-1 所示代码展示了一个二维示例。图 28-2 展示了一个 8*8 二维网格的 Morton 排序方法,其中包括 4 个 4*4 子块。在

行优先格式下，第一行的元素存储在第 0 ~ 7 个位置上，第二行的元素存储在第 8 ~ 15 个位置上，依次类推。但是在 Morton 排序中，能够有效保持二维数据的局部性（即二维空间中的邻居存储在一起）。

```
//split x leaving a zero bit between each original bit
int dilate_1(int x){
    x = (x ^ (x << 8)) & 0x00ff00ff;
    x = (x ^ (x << 4)) & 0x0f0f0f0f;
    x = (x ^ (x << 2)) & 0x33333333;
    x = (x ^ (x << 1)) & 0x55555555;
    return x;
}

//interleave row and column bits
int zindex2d (int column, int row) {
    return (dilate_1(row)<<1)|dilate_1(column);
}
```

图 28-1 从行、列索引计算 Morton 索引

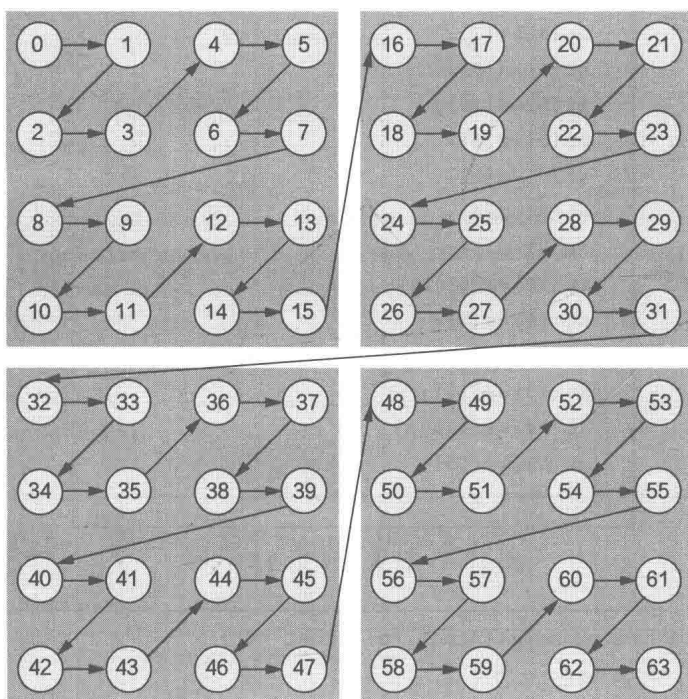


图 28-2 8*8 二维网格的 Morton 排序，细分为 4 个 4*4 子块

另一个关于 4*4 子块的特性是，若每个元素都是单精度浮点数据（4 字节），则这 16 个浮点数据正好可以存放在处理器的一个缓存行中，或者是协处理器的向量寄存器中，并可以通过一次缓存读完成数据加载。

28.3 矩阵转置

矩阵转置即实现 $A'[i, j] = A[j, i]$ ，其定义十分简单，图 28-3 展示了最简单的实现代码。

这一简单版本能够正确实现转置功能，但是由于在内层循环中频繁变换索引位置，因此在内存中非顺序地访问数据，同时我们也并不清楚内存的对齐情况。编译器也不会对该循环进行向量化。

使用 Morton 排序会极大地提升性能。考虑图 28-2 中的 4 个 4*4 子矩阵，我们知道实现矩阵转置可以通过首先转置每个子矩阵中的元素然后再对 4 个子矩阵整体做转置来实现。由于每个子矩阵可以存放在一个缓存行中，因此程序只需要 4 次缓存读操作就能加载整个 8*8 矩阵中的所有 64 个元素。转置一个 4*4 矩阵值需要重排一个缓存行中的 16 个数据，并且这一重排方法对所有 4 个子矩阵是相同的。图 28-4 展示了一个 4*4 转置的可移植实现方式。Intel Xeon Phi 协处理器可以通过一个 `_mm512_permutevar_epi32` 指令来实现这一转置，如图 28-5 所示。

```
float a[N*N], at[N*N];
for(i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        at[j*N+i] = a[i*N+j];
    }
}
```

图 28-3 矩阵转置简单的实现代码

```
void transpose4x4(float *a, float *at) {
    at[0]=a[0];
    at[1]=a[2];
    at[2]=a[1];
    at[3]=a[3];
    at[4]=a[8];
    at[5]=a[10];
    at[6]=a[9];
    at[7]=a[11];
    at[8]=a[4];
    at[9]=a[6];
    at[10]=a[5];
    at[11]=a[7];
    at[12]=a[12];
    at[13]=a[14];
    at[14]=a[13];
    at[15]=a[15];
}
```

图 28-4 转置 4*4 矩阵的代码

```
void transpose4x4(float *a, float *at) {
    __m512 oa;
    __m512 ta;
    __m512i pat={0,2,1,3,8,10,9,11,4,6,5,7,12,14,13,15};
    oa=_mm512_load_ps(a);
    ta=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pat,_mm512_castps_si512(oa))
    );
    _mm512_store_ps(at,ta);
}
```

图 28-5 利用 Intel Xeon 内置函数转置 4*4 矩阵代码

注意 为了保证索引的简洁性, 该代码要求矩阵大小是 4 的倍数并且存储空间大小是 2 的幂的平方。一般情况下可以放松这一限制, 但是本章忽略加边 (padding) 以及边角处理等方法。

利用较大规模的矩阵比较基于 Morton 排序 (图 28-6) 和简单实现的矩阵转置 (图 28-7) 性能, 可以看到在 Intel Xeon Phi 协处理器上 (见图 28-8) 对于大于 1024×1024 的矩阵规模, 单线程加速比为 9; 在 244 个线程的情况下, 在矩阵规模小于 16384×16384 的情况下 Morton 排序并无优势。

```
int main(int argc, char **argv) {
    int i,j,k,iad;
    int uad,lad;
    int N=8;
    int N4=N/4;
    float *a=(float *)_mm_malloc(N*N*sizeof(float),128);
    float *c=(float *)_mm_malloc(N*N*sizeof(float),128);
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            iad=zindex2d(j,i);
            a[iad]=j+i*N; //dummy data
        }
    }
    for(i=0; i<N4; i++) {
        for(j=0; j<N4; j++) {
            uad=zindex2d(j,i); //offset of original submatrix
            lad=zindex2d(i,j); //offset of transpose submatrix
            transpose4x4(&a[16*uad],&c[16*lad]);
        }
    }
    _mm_free(a);
    _mm_free(c);
}
```

图 28-6 利用 Morton 排序转置 8×8 矩阵

```
int main(int argc, char **argv) {
    int i,j,k,iad;
    int uad,lad;
    int N=8;
    if(argc>1) {
        N=atoi(argv[1]);
        If(N>32768) N=32768; // limit size
    }
    // for now, N must be a multiple of 4
}
```

图 28-7 利用 Morton 排序和 OpenMP 多线程实现一般矩阵转置的代码


```
i=N/4;
if(N>(i*4)) N=(i+1)*4;
int N4=N/4;
// for now, matrix size power of 2
double l2=log(N)/log(2.0);
int il2=(int)l2;
if(il2<l2) il2++;
int MSZ=pow(2,il2);
printf("N=%d; N/4=%d; msize=%d\n",N,N4,MSZ);
float *a=
    (float *)_mm_malloc(MSZ*MSZ*sizeof(float),128);
float *c=
    (float *)_mm_malloc(MSZ*MSZ*sizeof(float),128);
int iv=0;
// initialize faster as long as we're threading
```

图 28-7 （续）

维度	简单实现 1个线程	Morton 1个线程	加速比	简单实现 244个线程	Morton 244个线程	加速比
256	0.0033	0.0014	2.46	0.0014	0.0037	0.38
512	0.0763	0.0066	11.60	0.0012	0.0155	0.07
1024	0.3304	0.0289	11.44	0.0114	0.0303	0.38
2048	1.2660	0.1499	8.44	0.0185	0.0476	0.39
4096	4.9433	0.5820	8.49	0.0359	0.0669	0.53
8192	20.1280	2.2590	8.91	0.1156	0.1477	0.78
16384	80.2921	8.9273	8.99	0.4316	0.3451	1.25
32768	328.4720	35.7564	9.18	1.8124	0.7410	2.44

图 28-8 Intel Xeon Phi 协处理器上单线程和多线程的简单矩阵转置以及基于 Morton 排序的矩阵转置性能，第 2、3、5、6 列的单位为秒

图 28-9 展示了 Intel Xeon 处理器性能结果，单线程情况下，对于大小为 32768*32768 的矩阵达到 2.6 倍加速；32 个线程的情况下，对大部分规模的矩阵都有性能提升，最大加速比为 4。

维度	简单实现 1个线程	Morton 1个线程	加速比	简单实现 32个线程	Morton 32个线程	加速比
256	0.0004	0.0003	1.52	0.0014	0.0037	1.73
512	0.0021	0.0009	2.26	0.0012	0.0155	1.00
1024	0.0067	0.0061	1.09	0.0114	0.0303	1.59
2048	0.0147	0.0282	0.52	0.0185	0.0476	0.59
4096	0.2043	0.1757	1.16	0.0359	0.0669	0.67
8192	0.8254	0.5219	1.58	0.1156	0.1477	2.30
16384	3.7540	1.7044	2.20	0.4316	0.3451	3.39
32768	18.4287	6.8835	2.68	1.8124	0.7410	4.07

图 28-9 Intel Xeon 处理器上单线程和多线程的简单矩阵转置以及基于 Morton 排序的矩阵转置性能，第 2、3、5、6 列的单位为秒

28.4 矩阵乘法

矩阵乘法与矩阵转置算法十分相关，前者需要对第一个矩阵每行与第二矩阵每列的乘积进行求和。

给定一个大小为 $M \times L$ 的矩阵 A 和一个大小为 $L \times N$ 的矩阵 B ，可以通过下面的代码实现矩阵乘法 $C = A \times B$ ：

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++){
        sum=0;
        for (k=0; k<L; k++)
            sum += A[i,k]*B[k,j];
        C[i,j] = A[i,k]*B[k,j];
    }
```

这种简单的实现代码几乎无法利用数据局部性。通常提高性能的方法是利用某种分块方法，因此可以考虑利用在前面矩阵转置中用到的 Morton 排序方法。考虑矩阵 A 和 B 的一个 4×4 分块，可以看到（见图 28-10）我们可以利用两个子块计算矩阵 C 的部分元素的部分数值。由于每个分块可以存储在一个缓存行中，因此数据访问十分高效。此时程序需要某种索引位置的模式，但是这种模式对矩阵 A 和 B 的每对子矩阵都是一样的。

图 28-11 展示了利用 Intel Xeon Phi 协处理器内置函数实现 4×4 矩阵子块乘法的代码。

在该 4×4 矩阵子块乘算法的外围，只需要增加一些循环使之遍历矩阵 A 和矩阵 B 的所有子块对，并对矩阵 C 4×4 子块的计算结果进行累加。为了简化程序，再次假设矩阵大小是 2 的幂，并且是 4 的倍数。由于每个 4×4 子块的结果可以独立计算，因此类似矩阵转置的

代码, 我们利用 OpenMP 在循环外层进行并行化。图 28-12 展示了最终的程序代码。注意, `#ifdef __MIC__` 宏选择 Intel Xeon Phi 协处理器的内置函数, 或者使用普通的 C 语言表达式。

```
void zmatmul(float *a, float *b, float *c){
    c[0] += a[0] * b[0] + a[1] * b[2] + a[4] * b[8] + a[5] * b[10];
    c[1] += a[0] * b[1] + a[1] * b[3] + a[4] * b[9] + a[5] * b[11];
    c[4] += a[0] * b[4] + a[1] * b[6] + a[4] * b[12] + a[5] * b[14];
    c[5] += a[0] * b[5] + a[1] * b[7] + a[4] * b[13] + a[5] * b[15];
    c[2] += a[2] * b[0] + a[3] * b[2] + a[6] * b[8] + a[7] * b[10];
    c[3] += a[2] * b[1] + a[3] * b[3] + a[6] * b[9] + a[7] * b[11];
    c[6] += a[2] * b[4] + a[3] * b[6] + a[6] * b[12] + a[7] * b[14];
    c[7] += a[2] * b[5] + a[3] * b[7] + a[6] * b[13] + a[7] * b[15];
    c[8] += a[8] * b[0] + a[9] * b[2] + a[12] * b[8] + a[13] * b[10];
    c[9] += a[8] * b[1] + a[9] * b[3] + a[12] * b[9] + a[13] * b[11];
    c[12] += a[8] * b[4] + a[9] * b[6] + a[12] * b[12] + a[13] * b[14];
    c[13] += a[8] * b[5] + a[9] * b[7] + a[12] * b[13] + a[13] * b[15];
    c[10] += a[10] * b[0] + a[11] * b[2] + a[14] * b[8] + a[15] * b[10];
    c[11] += a[10] * b[1] + a[11] * b[3] + a[14] * b[9] + a[15] * b[11];
    c[14] += a[10] * b[4] + a[11] * b[6] + a[14] * b[12] + a[15] * b[14];
    c[15] += a[10] * b[5] + a[11] * b[7] + a[14] * b[13] + a[15] * b[15];
}
```

图 28-10 利用普通乘法和加法指令实现大小为 24*4 的矩阵乘

```
void zmatmul(float *ain, float *bin, float *cout) {
    __m512 a,b,c;
    __m512 a0,a1,a2,a3;
    __m512 b0,b1,b2,b3;
    __m512i pa0={ 0, 0, 2, 2, 0, 0, 2, 2, 8, 8, 10, 10, 8, 8, 10, 10};
    __m512i pa1={ 1, 1, 3, 3, 1, 1, 3, 3, 9, 9, 11, 11, 9, 9, 11, 11};
    __m512i pa2={ 4, 4, 6, 6, 4, 4, 6, 6, 12, 12, 14, 14, 12, 12, 14, 14};
    __m512i pa3={ 5, 5, 7, 7, 5, 5, 7, 7, 13, 13, 15, 15, 13, 13, 15, 15};

    __m512i pb0={ 0, 1, 0, 1, 4, 5, 4, 5, 0, 1, 0, 1, 4, 5, 4, 5};
    __m512i pb1={ 2, 3, 2, 3, 6, 7, 6, 7, 2, 3, 2, 3, 6, 7, 6, 7};
    __m512i pb2={ 8, 9, 8, 9, 12, 13, 12, 13, 8, 9, 8, 9, 12, 13, 12, 13};
    __m512i pb3={ 10, 11, 10, 11, 14, 15, 14, 15, 10, 11, 10, 11, 14, 15, 14, 15};

    a=_mm512_load_ps(ain);
    b=_mm512_load_ps(bin);
    c=_mm512_load_ps(cout);

    a0=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa0,_mm512_castps_si512(a)));
    a1=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa1,_mm512_castps_si512(a)));
    a2=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa2,_mm512_castps_si512(a)));
    a3=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa3,_mm512_castps_si512(a)));

    b0=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb0,_mm512_castps_si512(b)));
    b1=_mm512_castsi512_ps(
```

图 28-11 在 Intel Xeon Phi 协处理器上利用内置函数乘法和加法指令实现矩阵乘


```

    _mm512_permutevar_epi32(pb1,_mm512_castps_si512(b)));
    b2=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb2,_mm512_castps_si512(b)));
    b3=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb3,_mm512_castps_si512(b)));

    c=_mm512_fmadd_ps(a0,b0,c);
    c=_mm512_fmadd_ps(a1,b1,c);
    c=_mm512_fmadd_ps(a2,b2,c);
    c=_mm512_fmadd_ps(a3,b3,c);

    _mm512_store_ps(cout,c);
}

```

图 28-11 (续)

```

void zmatmul(float *ain, float *bin, float *cout) {

#ifdef _MIC_ //use intrinsics if an Intel Xeon Phi coprocessor
    _mm512 a,b,c;
    _mm512 a0,a1,a2,a3;
    _mm512 b0,b1,b2,b3;
    _m512i pa0={ 0, 0, 2, 2, 0, 0, 2, 2, 8, 8,10,10, 8, 8,10,10};
    _m512i pa1={ 1, 1, 3, 3, 1, 1, 3, 3, 9, 9,11,11, 9, 9,11,11};
    _m512i pa2={ 4, 4, 6, 6, 4, 4, 6, 6,12,12,14,14,12,12,14,14};
    _m512i pa3={ 5, 5, 7, 7, 5, 5, 7, 7,13,13,15,15,13,13,15,15};
    _m512i pb0={ 0, 1, 0, 1, 4, 5, 4, 5, 0, 1, 0, 1, 4, 5, 4, 5};
    _m512i pb1={ 2, 3, 2, 3, 6, 7, 6, 7, 2, 3, 2, 3, 6, 7, 6, 7};
    _m512i pb2={ 8, 9, 8, 9,12,13,12,13, 8, 9, 8, 9,12,13,12,13};
    _m512i pb3={10,11,10,11,14,15,14,15,10,11,10,11,14,15,14,15};

    a=_mm512_load_ps(ain);
    b=_mm512_load_ps(bin);
    c=_mm512_load_ps(cout);

    a0=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa0,_mm512_castps_si512(a)));
    a1=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa1,_mm512_castps_si512(a)));
    a2=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa2,_mm512_castps_si512(a)));
    a3=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pa3,_mm512_castps_si512(a)));

    b0=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb0,_mm512_castps_si512(b)));
    b1=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb1,_mm512_castps_si512(b)));
    b2=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb2,_mm512_castps_si512(b)));
    b3=_mm512_castsi512_ps(
        _mm512_permutevar_epi32(pb3,_mm512_castps_si512(b)));

    c=_mm512_fmadd_ps(a0,b0,c);
    c=_mm512_fmadd_ps(a1,b1,c);
    c=_mm512_fmadd_ps(a2,b2,c);
    c=_mm512_fmadd_ps(a3,b3,c);

    _mm512_store_ps(cout,c);
#else

```

图 28-12 完整的矩阵乘法代码


```

cout[0]+=ain[0]*bin[0]+ain[1]*bin[2]+ain[4]*bin[8]+ain[5]*bin[10];
cout[1]+=ain[0]*bin[1]+ain[1]*bin[3]+ain[4]*bin[9]+ain[5]*bin[11];
cout[2]+=ain[2]*bin[0]+ain[3]*bin[2]+ain[6]*bin[8]+ain[7]*bin[10];
cout[3]+=ain[2]*bin[1]+ain[3]*bin[3]+ain[6]*bin[9]+ain[7]*bin[11];
cout[4]+=ain[0]*bin[4]+ain[1]*bin[6]+ain[4]*bin[12]+ain[5]*bin[14];
cout[5]+=ain[0]*bin[5]+ain[1]*bin[7]+ain[4]*bin[13]+ain[5]*bin[15];
cout[6]+=ain[2]*bin[4]+ain[3]*bin[6]+ain[6]*bin[12]+ain[7]*bin[14];
cout[7]+=ain[2]*bin[5]+ain[3]*bin[7]+ain[6]*bin[13]+ain[7]*bin[15];
cout[8]+=ain[8]*bin[0]+ain[9]*bin[2]+ain[12]*bin[8]+ain[13]*bin[10];
cout[9]+=ain[8]*bin[1]+ain[9]*bin[3]+ain[12]*bin[9]+ain[13]*bin[11];
cout[10]+=ain[10]*bin[0]+ain[11]*bin[2]+ain[14]*bin[8]+ain[15]*bin[10];
cout[11]+=ain[10]*bin[1]+ain[11]*bin[3]+ain[14]*bin[9]+ain[15]*bin[11];
cout[12]+=ain[8]*bin[4]+ain[9]*bin[6]+ain[12]*bin[12]+ain[13]*bin[14];
cout[13]+=ain[8]*bin[5]+ain[9]*bin[7]+ain[12]*bin[13]+ain[13]*bin[15];
cout[14]+=ain[10]*bin[4]+ain[11]*bin[6]+ain[14]*bin[12]+ain[15]*bin[14];
cout[15]+=ain[10]*bin[5]+ain[11]*bin[7]+ain[14]*bin[13]+ain[15]*bin[15];
#endif
}

int main(int argc, char **argv) {
    int i,j,k,iad;
    int aad,bad,cad;
    int N=16;

    if(argc>1) {
        N=atoi(argv[1]);
        if(N>32768) N=32768; // max dim for now
    }

    i=N/4;
    if(N>(i*4)) N=(i+1)*4;

    double l2=log(N)/log(2.0);
    int il2=(int)l2;
    if(il2<l2) il2++;
    int MSZ=pow(2,il2);
    int N4=N/4;

    printf("N=%d; N/4=%d; msize=%d\n",N,N4,MSZ);

    // actual memory allocation must be power of 2 although N can be any // multiple of 4
    float *a=(float *)_mm_malloc(MSZ*MSZ*sizeof(float),128);
    float *b=(float *)_mm_malloc(MSZ*MSZ*sizeof(float),128);
    float *c=(float *)_mm_malloc(MSZ*MSZ*sizeof(float),128);

    int iv=0;
    #pragma omp parallel for private (i,j,iv,iad)
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            iv=j+i*N;
            iad=zorder2d_c2i(j,i);
            a[iad]=b[iad]=iv; //dummy data
            c[iad]=0; //initialize output array
        }
    }

    for(i=0; i<N4; i++) {
        #pragma omp parallel for private (aad,bad,cad,i,j,k)
        for(j=0; j<N4; j++) {
            cad=16*zorder2d_c2i(j,i); //offset for result block
            for(k=0; k<N4; k++) {
                aad=16*zorder2d_c2i(k,i); //offset for a block
                bad=16*zorder2d_c2i(j,k); //offset for b block
            }
        }
    }
}

```

图 28-12 (续)


```
        zmatmul(&a[aad],&b[bad],&c[cad]);
    }
}
}

_mm_free(a);
_mm_free(b);
_mm_free(c);

}
```

图 28-12 （续）

在 Intel Xeon Phi 协处理器上基于 Morton 的代码的性能结果如图 28-13 所示，矩阵维度范围从 256 到 32768。与简单实现对比，对于所有矩阵大小具有较大的性能提升，对 512*512 以及更大的矩阵，加速比超过 30。

图 28-14 展示了 Intel Xeon 处理器上的性能结果，基于 Morton 排序的矩阵乘法性能在矩阵规模大于 1024*1024 时超过简单矩阵乘法算法，并且在更大规模情况下可达到接近 30 倍的加速比。

维度	简单实现 244个线程	Morton 244个线程	加速比
256	0.02	0.003	6.67
512	0.26	0.008	32.5
1024	1.78	0.046	38.7
2048	12.87	0.4	32.18
4096	105.5	2.9	36.38
8192	105.5	23	36.74
16384	6597	181	36.4
32768	Too long	1468	--

图 28-13 在 Intel Xeon Phi 协处理器上 244 个线程的简单矩阵乘法以及基于 Morton 排序的矩阵乘法性能，第 2、3 列的单位为秒

维度	简单实现 32个线程	Morton 32个线程	加速比
256	0.1	0.188	0.53
512	0.05	0.169	0.29
1024	0.62	0.366	1.7
2048	2.89	0.871	3.3
4096	211	7.92	26.6
8192	1850	60.2	30.7
16384	13695	473	29.0
32768	Too long	3989	--

图 28-14 在 Intel Xeon 处理器上 32 个线程的简单矩阵乘法以及基于 Morton 排序的矩阵乘法性能，第 2、3 列的单位为秒

28.5 总结

本章介绍了另一种 Morton 排序方法，它可以作为常用的行优先格式和列优先格式多维

数据序列化方法的补充,能够有效保证数据局部性。我们给出了矩阵转置和矩阵乘法两个应用示例,实验结果证明了该方法在 Intel Xeon Phi 协处理器以及 Intel Xeon 处理器上的高效性。我们限定了矩阵大小为 2 的幂,并且只处理单精度浮点数据,但是可以利用其他技术处理一般矩阵,在这里主要介绍 Morton 排序对性能的影响。

与简单矩阵转置算法相比,对于矩阵规模大于 256×256 的矩阵,基于 Morton 排序的矩阵转置算法在 Intel Xeon Phi 协处理器上的单线程性能加速比可达 9 倍。在高度并行的实现中,例如在 244 个线程的情况下,基于 Morton 排序的算法可达 2 倍加速比。同样的实现在 Intel Xeon 处理器上单线程可达 2 倍加速比,在 32 个线程的情况下可达 4 倍加速比。

矩阵乘法能够获得更大的性能提升,对于规模大于 512×512 的矩阵而言,加速比从 30 到 40 不等。由于矩阵乘法的计算密集度要高于矩阵转置,因此 Morton 能更大幅度地提升代码性能。

基于 Morton 排序的算法在 Intel Xeon Phi 协处理器以及 Intel Xeon 处理器上均能提高程序性能,特别是对规模较大的矩阵,这均归功于更好的数据局部性。由于处理器具备乱序执行能力,因此它们可以有效隐藏延迟。当前协处理器是顺序执行的,因此它们对内存访问延迟更为敏感。但是,对于规模较大的矩阵而言,两个平台上的程序性能均受制于缓存和 TLB(translation look-aside buffer, 旁路转换缓冲区)缺失,因此按照 Morton 排序重组数据,使得一个缓存行的数据均能高效利用,可以有效降低访问缺失率。

其他数据结构也可借鉴 Morton 或者其他能够保持局部性的数据映射方法,例如几何模型,光栅数据,其他离散图像数据(基于像素的图像、视频、动画、地形、地震数据),坐标,数据库表,统计数据以及其他数据,并且该方法可以应用到更高维数据中,而不仅仅是二维数据。

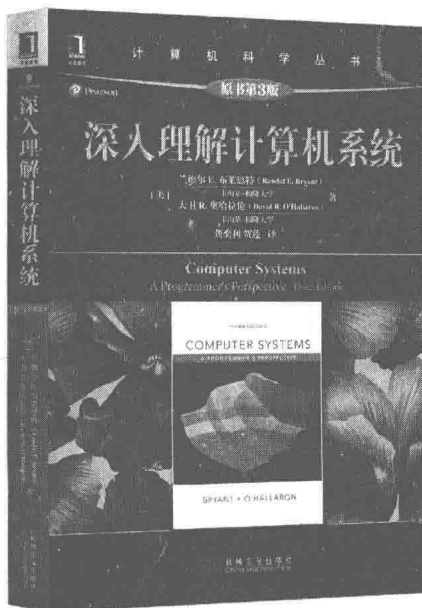
如前所述,在利用空间填充曲线和其他缓存局部性优化方法方面已经存在大量研究工作,本章最后提供一些优秀的参考文献。

28.6 更多信息

下面是一些与本章所述研究相关的文献。

- Bader, M., 2013. Space-Filling Curves—An Introduction with Applications in Scientific Computing. Springer.
- Chatterjee, S., Sen, S., 2000. Cache-Efficient Matrix Transposition. In: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA-6), pp. 195-205.
- Valsalam, V., Skjellum, A., 2002. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurr. Comput.* 14(10), 805-839.
- 顺序曲线, http://en.wikipedia.org/wiki/Z-order_curve。
- 解码 Morton 代码, <http://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/>。
- 本章及其他章代码下载地址 <http://lotsofcores.com>。

推荐阅读



深入理解计算机系统（原书第3版）

作者：[美] 兰德尔 E. 布莱恩特 等 译者：龚奕利 等 书号：978-7-111-54493-7 定价：139.00元

理解计算机系统首选书目，10余万程序员的选择

卡内基-梅隆大学、北京大学、清华大学、上海交通大学等国内外众多知名高校选用指定教材

从程序员视角全面剖析的实现细节，使读者深刻理解程序的行为，将所有计算机系统的相关知识融会贯通
新版本全面基于X86-64位处理器

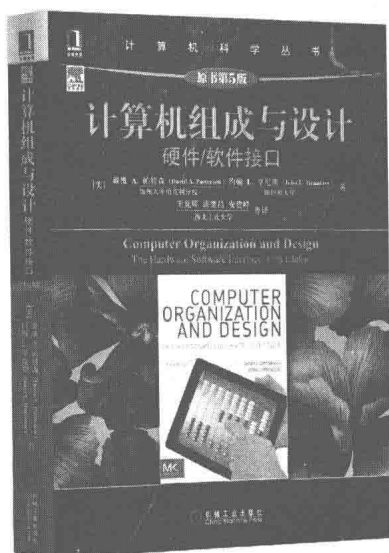
基于该教材的北大“计算机系统导论”课程实施已有五年，得到了学生的广泛赞誉，学生们通过这门课程的学习建立了完整的计算机系统的知识体系和整体知识框架，养成了良好的编程习惯并获得了编写高性能、可移植和健壮的程序的能力，奠定了后续学习操作系统、编译、计算机体系结构等专业课程的基础。北大的教学实践表明，这是一本值得推荐采用的好教材。本书第3版采用最新x86-64架构来贯穿各部分知识。我相信，该书的出版将有助于国内计算机系统教学的进一步改进，为培养从事系统级创新的计算机人才奠定很好的基础。

——梅宏 中国科学院院士/发展中国家科学院院士

以低年级开设“深入理解计算机系统”课程为基础，我先后在复旦大学和上海交通大学软件学院主导了激进的教学改革……现在我课题组的青年教师全部是首批经历此教学改革的学生。本科的扎实基础为他们从事系统软件的研究打下了良好的基础……师资力量的补充又为推进更加激进的教学改革创造了条件。

——臧斌宇 上海交通大学软件学院院长

推荐阅读



计算机组成与设计：硬件/软件接口（原书第5版）

作者：[美] 戴维 A. 帕特森 等 ISBN：978-7-111-50482-5 定价：99.00元

本书是计算机组成与设计的经典畅销教材，第5版经过全面更新，关注后PC时代发生在计算机体系结构领域的革命性变革——从单核处理器到多核微处理器，从串行到并行。本书特别关注移动计算和云计算，通过平板电脑、云体系结构以及ARM（移动计算设备）和x86（云计算）体系结构来探索和揭示这场技术变革。

计算机体系结构：量化研究方法（英文版·第5版）

作者：[美] John L. Hennessy 等 ISBN：978-7-111-36458-0 定价：138.00元

本书系统地介绍了计算机系统的设计基础、指令集系统结构，流水线和指令集并行技术。层次化存储系统与存储设备。互连网络以及多处理器系统等重要内容。在这个最新版中，作者更新了单核处理器到多核处理器的历史发展过程的相关内容，同时依然使用他们广受好评的“量化研究方法”进行计算设计，并展示了多种可以实现并行，隧的技术，而这些技术可以看成是展现多处理器体系结构威力的关键！在介绍多处理器时，作者不但讲解了处理器的性能，还介绍了有关的设计要素，包括能力、可靠性、可用性和可信性。

高性能并行珠玑 多核和众核编程方法

High Performance Parallelism Pearls Multicore and Many-core Programming Approaches

本书将使为Intel Xeon Phi产品开发高层并行性（包括最优编程）更加简单。Intel Xeon和Intel Xeon Phi系列之间的通用编程方法对整个科学和工程领域都是有利的，相同的程序可以实现多核和众核的并行可扩展性和向量化。

—— 选自推荐序, Sverre Jarp, CERN

本书展示了如何借助同种编程方法来利用处理器和协处理器上的并行性，展示了如何有效地利用配备Intel Xeon Phi协处理器和Intel Xeon处理器或者其他多核处理器的系统的能力。本书包括多个行业和领域（如化学、工程以及环境科学）中成功的编程示例。每一章都包括所使用的编程技巧的详细讲解，并且展示了在Intel Xeon Phi协处理器和多核处理器上的高性能结果。这些示例不仅展示了这些高性能系统的特征，还给出了利用这些新型异构系统间并行性的方法。

本书特色

- 推动一致的基于标准的编程，展示多核处理器和Intel Xeon Phi协处理器上高性能编码的细节。
- 涵盖多个垂直领域的示例，展示真实应用代码的并行优化。
- 源代码可供下载，以便于未来进一步研究。

作者简介

詹姆斯·赖因德斯（James Reinders）Intel公司软件总监，首席技术布道师。参与多个旨在加强并行编程在工业界应用的工程研究和教育项目。他对多个项目做出了贡献，包括世界上首例TFLOPS级超级计算机（ASCI Red）和世界上首例TFLOPS级微处理器（Intel Xeon Phi协处理器）。

吉姆·杰弗斯（Jim Jeffers）Intel公司技术计算组的首席工程师和工程经理。目前，Jim是Intel技术计算可视化团队的领导。

本书译自原版 *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*
并由Elsevier授权出版



投稿热线: (010) 88379604
客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/高性能计算

ISBN 978-7-111-58080-5



9 787111 580805 >

定价: 119.00元